1-14-2019

# Accelerating Reactive-Flow Simulations via Vectorized Chemical Kinetic Evaluation

Nicholas Curtis
*University of Connecticut - Storrs*, nicholas.curtis@uconn.edu

# Accelerating Reactive-Flow Simulations via Vectorized Chemical Kinetic Evaluation

Nicholas Curtis, Ph.D.

University of Connecticut, 2018

This work details efforts to reduce the cost of using detailed chemical kinetic modeling in realistic reactive-flow simulations, utilizing analytical Jacobian evaluation and vectorized-computing on the central processing unit (CPU), graphics processing unit (GPU) and other hardware-accelerators.

The first part of this thesis investigated GPU-based ordinary differential equation (ODE) methods for stiff chemical kinetics. A fifth-order implicit Runge–Kutta method and two fourth-order exponential integration methods were implemented for the GPU and paired with the analytical chemical kinetic Jacobian software `pyJac`. The performance of each algorithm was compared with a commonly used CPU-based implicit integrator `CVODE`. The implicit Runge–Kutta method running on a single Tesla C2075 GPU was equivalent to `CVODE` running on 12–38 CPU cores for integration of hydrogen and methane kinetic models using a smaller global integration time-step, however the performance of the GPU-solver degraded at a larger time-steps due to thread divergence and higher memory traffic.

The second part of this work investigated the performance of vectorized evaluation of constant-pressure/volume thermochemical source-term and sparse/dense chemical kinetic Jacobians using single-instruction, multiple-data (SIMD) and single-instruction, multiple thread (SIMT) paradigms; the developed codes were additionally incorporated into `pyJac`. A new formulation of the chemical kinetic governing equations was derived and verified, resulting in greatly increased Jacobian sparsities. Significant speedups were found for shallow-vectorized OpenCL source-rate evaluation as compared with a parallel OpenMP code, increasing for sparse and dense chemical kinetic Jacobian evaluation. Further, the developed work was shown to be orders of magnitude faster than a simple first-order finite-difference Jacobian approach.

Finally, several CPU-vectorized linearly-implicit Rosenbrock solvers were adapted for use with `pyJac`, and validated against `CVODE`. The open-source computational fluid dynamics code `OpenFOAM` was extended to utilize the vectorized solvers, and the eddy dissipation concept combustion model was adapted for their use. The `OpenFOAM`-coupled vectorized solver was validated over a range of zero-dimensional homogeneous ignition problem against `Cantera`, before its performance and precision were compared to built-in `OpenFOAM` solvers for a case modeling the Sandia Flame D; a speedup of 12–15 $\times$ was found for the vectorized solver.

# Accelerating Reactive-Flow Simulations via Vectorized Chemical Kinetic Evaluation

Nicholas Curtis

B. Eng., McGill University, 2012

M. Sci., University of Connecticut, 2014

A Disseration

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut 2019

APPROVAL PAGE

Doctor of Philosophy Dissertation

Accelerating Reactive-Flow Simulations via Vectorized Chemical Kinetic Evaluation

Presented by

Nicholas Curtis, B.Eng., M.A.

Major Advisor _____
Dr. Chih-Jen Sung

Associate Advisor _____
Dr. Tianfieng Lu

Associate Advisor _____
Dr. Baki Cetegen

Associate Advisor _____
Dr. Bryan Weber

Associate Advisor _____
Dr. Xinyu Zhao

University of Connecticut
2019

*To my Grandmother, who I love with all my heart. I know just how proud of me you'd be.*

# Acknowledgements

There are a number of people and organizations without whom this dissertation would not have happened.

I would first like to thank my advisor, Dr. Chih-Jen Sung, for his support of my work, allowing me the independence to pursue topics I found most interesting, his patience when I found myself struggling with the joys of buggy vectorization platforms, and his excellent guidance along the way. I also would like to thank the members of my committee, Dr. Baki Cetegen, Dr. Tianfeng Lu, Dr. Bryan Weber and Dr. Xinyu Zhao for their efforts reviewing this document and constructive reviews of my work.

Next, I owe a debt to current and past colleagues at the University of Connecticut, including Dr. Kyle Brady, Dr. Bryan Weber, Dr. Xin Xue, Dr. Goutham Kukkadapu, Dr. Kamal Kumar, Dr. Pradeep Singh, Ruozhou Fang, Xiao Ren, Mengyuan Wang, Kyle Twarog, and Peter Vannorsdall. You each made this process better in your own way, by giving advice, helping me out when needed, and improving the work-day with good conversation. In addition, I must give special thanks to Dr. Kyle Niemeyer of Oregon State University for our many collaborations, discussions and joint efforts which form a large portion of this work. Further, I would like to thank Dr. Christopher Stone of Computational Science and Engineering, LLC. for graciously allowing me to use his vectorized integration methods for this effort, and Heikki Kahila of Aalto University for our many conversations about `OpenFOAM` and computational fluid dynamics.

I must also thank the National Science Foundation for supporting this work (under grant ACI-1534688) and allowing me the freedom to pursue interesting and meaningful research.

Further, I would like to thank my parents and brother Timothy, who from a young age kindled my curiosity in all things scientific, mathematical and provided me with more books than I could dream of. You should be so lucky as to play “Death by Numbers” at the table when your family goes out to eat.

I must also thank my friend Michael Sanchick who helped me through some of the hardest times of my life, and has been there for many of the best ones.

Finally, I have to thank Bre for supporting me over this last year and a half, and giving me joy in

my life, even when I'm at the lab until two in the morning night after night. I would not have survived this past summer without you, and I can't wait to see where we go next.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

The combustion of fossil fuels accounted for over 80 % of the total energy consumption in the United States in 2017 [1], and is projected to remain the country's dominant source of energy production through the year 2050 [2]. At the same time, there has been a global push to reduce overall emissions levels to meet increasingly stringent efficiency and pollutant standards, and help mitigate the effects of climate change. In order to meet these goals while still satisfying global energy demand, the development of next-generation combustion devices has focused on improved fuel efficiency, emissions reductions and fuel flexibility to accommodate new alternative (non-petroleum-based) fuels. In turn, this has driven the consideration of new combustion modes—such as low-temperature combustion (LTC) [3] or mild to intense low oxygen dilution (MILD) [4] combustion—that approach the limits of operating stability. Computational combustion modeling has proved to be an important tool in this effort, aiding, for instance, in the rapid development of new engine concepts such as the homogeneous charge compression ignition (HCCI) engine [5].

In HCCI combustion, low concentrations of fuel are injected early in the engine cycle giving time for the air-fuel mixture to become homogeneous; the mixture is then highly compressed to autoignite the fuel. This combustion mode has the potential to deliver high thermodynamic efficiencies, comparable to diesel engines, but with significant reductions in NOx and soot emissions [3]. However, as the ignition delay and burning rates are strongly dependent on the chemical kinetics it is difficult to control the phasing of combustion in HCCI engines [6], and therefore, in-order to achieve predictive computational modeling the use of accurate finite-rate chemistry is required. Similarly, predicting other combustion limit phenomena, e.g., local flame extinction—relevant to lean blow out in gas turbine engines—requires accurate chemical kinetic

models [7].

As the need for detailed and accurate chemical kinetic models in predictive reactive-flow simulations has become recognized, models describing the oxidation of hydrocarbon fuels simultaneously grew orders of magnitude in size and complexity. For example, detailed kinetic models consisting of thousands of species and tens of thousands of reactions exist for 2-methylalkanes (relevant for jet and diesel fuel surrogates) [8], gasoline [9, 10] and biodiesel [11]. Generally speaking, the computational cost of solving the associated chemical kinetic system of equations scales, at best, quadratically with the number of species in the model (and cubically at worst) [7], making use of such large models challenging even in simpler zero- or one-dimensional analyses. Several recent studies [12–14] demonstrated that using even modestly sized chemical kinetic models can incur severe computation cost for reactive-flow simulations. For example, a single high-resolution Large Eddy Simulation (LES) realization of a diesel spray—using up to 22 million grid cells with a 54-species *n*-dodecane model—for 2 ms after start of injection with the common implicit CVODE solver [15] took 48,000 CPU core hours and up to 20 days of wall clock time [14].

A host of techniques have been developed over the years, as reviewed by Lu and Law [16] as well as Turányi and Tomlin [17], in an attempt to reduce the size and computational demands of large detailed chemical kinetic models while retaining fidelity. Some of the most commonly used approaches include skeletal reduction methods that remove unimportant species and reactions [18–21], lumping of species that share similar thermochemical properties [22–24], and time-scale reduction methods that reduce numerical stiffness [25–28]. Reduction strategies often combine multiple methods a priori [29–31] or apply them dynamically during a simulation to achieve greater local savings [32–36]. Tabulation/interpolation methods [37] are also used to reduce computational costs, and are often applied in concert with reduction methods [38, 39].

In addition to reduction methods that modify or approximate the base chemical kinetic model, significant computational performance gains can be realized by improvements to the integration algorithms that solve the chemical kinetic ordinary differential equations (ODEs). Typically combustion codes rely on robust, high-order implicit integration algorithms based on backward differentiation formulae [40–43] to efficiently deal with the high levels of numerical stiffness exhibited by most chemical kinetic models. In order to solve the non-linear algebraic equations that arise in these techniques, the chemical kinetic Jacobian matrix must be evaluated and factorized, resulting in the previously mentioned quadratic and cubic scaling of computational cost with chemical kinetic model size. Use of an analytical formulation for the Jacobian matrix, rather than a simple finite difference approximation, can drop the cost of Jacobian evaluation to a linear dependence on the number of species in the model [16, 44].

Along with improvements in stiff implicit algorithms used for chemical kinetics, methods tailored for high-performance vector-processors offer another path to reduce the cost of detailed chemical kinetic integration. Central processing unit (CPU) clock speeds have increased regularly in the past—i.e., Moore's Law—but this trend has slowed somewhat recently due to power consumption and heat dissipation issues [45]. While multicore parallelism continues to improve CPU performance, the use of single-instruction multiple data (SIMD) and related single-instruction multiple thread (SIMT) processing has continued to improve floating-operation throughput. Graphics processing units (GPUs)—SIMT processors originally developed for graphics/video processing and display purposes—consist of hundreds to thousands of cores, compared to the tens of cores found on a typical CPU. Recognizing that the operator-split chemistry integration that forms the basis of many reactive-flow codes [46] is a good fit for the SIMT-acceleration, a number of recent studies [47–54] explored the use of SIMT processors to accelerate the integration of chemical kinetics in reactive-flow codes. More recently, SIMD-based vector processing, e.g., as present on modern CPUs, has begun to be used to accelerate the solution of numerically stiff systems of ODEs [55, 56], including for chemical kinetics [57].

## 1.2 Outline

The first part of this dissertation (Chapter 2) will focus on the use of GPU-based integration algorithms for the solution of stiff chemical kinetics. Following this, Chapter 3 discusses the implementation, validation, and performance of an analytical chemical kinetic Jacobian code for SIMD and SIMT processors. In Chapter 4 the developed vectorized chemical kinetic ODE integration and analytical Jacobian evaluation methods will be combined and applied to a realistic reactive flow simulation to demonstrate their performance and accuracy. Finally in Chapter 5, a summary of the achievements of this work will given, along with recommendations for future efforts.

# List of publications

At time of this writing, the final chapter of this thesis is in-preparation for publication, while Chapter 2 and Chapter 3 have been published in publications [4] and [5], respectively.


[1] N. J. Curtis, K. E. Niemeyer, and C.-J. Sung. "An automated target species selection method for dynamic adaptive chemistry simulations". In: *Combust. Flame* 162.4 (2015), pp. 1358–1374. DOI: 10.1016/j.combustflame.2014.11.004.

[2] O. S. Abianeh, N. Curtis, and C.-J. Sung. "Determination of modeled luminosity-based and pressure-based ignition delay times of turbulent spray combustion". In: *Int. J. Heat Mass Transf.* 103 (2016), pp. 1297–1312. ISSN: 0017-9310. DOI: https://doi.org/10.1016/j.ijheatmasstransfer.2016.06.067.

[3] K. E. Niemeyer, N. J. Curtis, and C.-J. Sung. "`pyJac`: analytical Jacobian generator for chemical kinetics". In: *Comput. Phys. Comm.* (Feb. 2017). ISSN: 0010-4655. DOI: 10.1016/j.cpc.2017.02.004.

[4] N. J. Curtis, K. E. Niemeyer, and C.-J. Sung. "An investigation of GPU-based stiff chemical kinetics integration methods". In: *Combustion and Flame* 179 (2017), pp. 312–324. ISSN: 0010-2180. DOI: 10.1016/j.combustflame.2017.02.005.

[5] N. J. Curtis, K. E. Niemeyer, and C.-J. Sung. "Using SIMD and SIMT vectorization to evaluate sparse chemical kinetic Jacobian matrices and thermochemical source terms". In: *Combustion and Flame* 198 (2018), pp. 186–204. ISSN: 0010-2180. DOI: 10.1016/j.combustflame.2018.09.008.

[6] N. J. Curtis et al. *Vectorized chemical kinetic integration in realistic reactive-flow simulations*. In preparation.

# Chapter 2

# GPU-based stiff chemical kinetics integration methods

## 2.1 Introduction

A SIMD instruction utilizes a vector processing unit to execute the same instruction on multiple pieces of data, e.g., performing multiple floating point multiplications concurrently. In contrast, a SIMT process achieves SIMD parallelism by having many threads execute the same instruction concurrently. Many different flavors of SIMD/SIMT processing exist:

- Modern CPUs have vector processing units capable of executing SIMD instructions (e.g., SSE, AVX2)
- GPUs feature hundreds to thousands of separate processing units, and utilize the SIMT model
- Intel's Xeon Phi co-processor has tens of (hyperthreaded) cores containing wide-vector units designed for SIMD execution, with each core capable of running multiple independent threads

Using the SIMD/SIMT parallelism model requires extra consideration to accelerate chemical kinetics integration.
This study used the NVIDIA CUDA framework [58, 59], hence the following discussion will use CUDA terminology; however, the concepts within are widely applicable to SIMT processing. The basic parallel function call on a GPU, termed a kernel, is broken up into a grid of thread blocks as seen in Fig. 2.1. A GPU consists of many streaming multiprocessors (SMs), each of which is assigned one or more thread blocks in the grid. The SMs further subdivide the blocks into groups

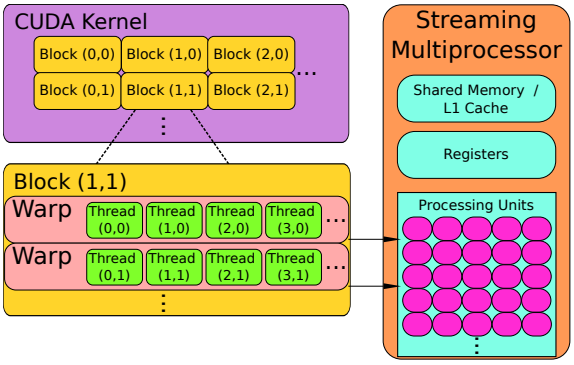



Figure 2.1: Example of the CUDA SIMT paradigm. Program calls (kernels) are split into a grid of blocks, which are in turn composed of threads. Threads are grouped in warps (note: warps are typically composed of 32 threads) and executed concurrently on streaming multiprocessors. Streaming multiprocessors have registers and L1 cache memory shared between all executing warps. Figure file is available under CC-BY [60].

of 32 threads called warps, which form the fundamental CUDA processing entity. The resources available on a SM (memory, processing units, registers, etc.) are split between the warps from all the assigned blocks. The threads in a warp are executed in parallel on CUDA cores (processing units), with multiple warps typically being executed concurrently on a SM. Thread divergence occurs when the threads in a warp follow different execution paths, e.g., due to if/then branching, and is a key performance concern for SIMT processing; in such cases the divergent execution paths must execute in serial. All threads in a warp are executed even if any thread in the warp is unfinished. When a divergent path is long and complicated or only a handful of threads in a warp require its execution, significant computational waste may occur as the other threads will be idle for long periods. A related concept of waste within a SIMD work unit is described by Stone et al. [57].

Furthermore, as compared with a typical CPU, GPUs possess relatively small memory caches and few registers per SM. These resources are further split between all the blocks/warps running on that SM (Fig. 2.1). Overuse of these resources can cause slow global memory accesses for data not stored locally in-cache or can even reduce the number of blocks assigned to each SM. The performance tradeoffs of various CUDA execution patterns are quite involved and beyond the scope of this work; for more details we refer the interested reader to several works that discussed these topics in depth [61–63]. Instead, we will briefly highlight key considerations for CUDA-based integration of chemical kinetic initial value problems (IVPs).

### 2.1.1 GPU-accelerated chemical kinetics

The extent of thread cooperation within a CUDA-based chemical kinetic IVP integration algorithm is a key point that shapes much of implementation. GPU-accelerated chemical kinetic solvers typically follow either a “per-thread” pattern [49, 51, 52], in which each individual GPU

thread solves a single chemical kinetic IVP, or a "per-block" approach [51, 53], in which all the threads in a block cooperate to solve the ordinary differential equations (ODEs) that comprise a single chemical kinetic IVP. The greatest potential benefit of a per-thread approach is that a much larger number of IVPs can theoretically be solved concurrently; the number of blocks that can be executed concurrently on each SM is usually around eight, whereas typical CUDA launch configurations in this work consist of 64 threads per block, or 512 sets of IVPs solved concurrently per SM. Unfortunately, the larger amount of parallelism offered by a per-thread approach comes with certain drawbacks. A per-thread approach may also encounter more cache-misses, since the memory available per SM must now be split between many more sets of IVPs. This results in expensive global memory loads. The performance of a per-thread approach can also be greatly impacted by thread divergence, because different threads may follow different execution paths within the IVP integration algorithm itself [51, 52]. For example, in a per-thread-based solver each thread in a warp advances its IVP by one internal integration step concurrently, and here on a step failure the thread simply does not update the solution vector at the end of the internal time-step. If only a handful of threads in a warp require many more internal time-steps than the others, they will force the majority of threads to wait until all threads in the warp have completed the global integration step, wasting computational resources. Additionally, implicit integration algorithms—which typically have complex branching and evaluation paths—may suffer more from thread divergence when implemented on a per-thread basis than relatively simpler explicit integration techniques [51]. The impact of thread divergence on integrators is typically less severe when following a per-block strategy, since the execution path of each thread is planned by design of the algorithm. A per-block approach also offers significantly more local cache memory and available registers for solving an IVP, and thus memory access speed and cache size are less of a concern. However, in our experience, optimizing use of these resources requires significant manual tuning and makes it more difficult to generalize the developed algorithm between different chemical kinetic models—a key feature for potential non-academic applications. In addition, Stone and Davis [51] showed that a per-thread implicit integration algorithm outperforms the per-block implementation of the same algorithm in the best-case scenario (elimination of thread divergence by choice of identical initial conditions). Various studies in recent years explored the use of high-performance SIMT devices to accelerate (turbulent) reactive-flow simulations. Spafford et al. [47] investigated a GPU implementation of a completely explicit—and thus well suited for SIMT-acceleration—direct numerical simulation code for compressible turbulent combustion. Using a Tesla C1060 GPU, an order of magnitude speedup was demonstrated for evaluation of species production rates compared to a sequential CPU implementation on a AMD-Operton processor; evaluating chemical source terms is much

simpler than chemical kinetics integration on GPUs. Shi et al. [48] used a Tesla C2050 GPU to evaluate species rates and factorize the Jacobian for the integration of (single) independent kinetics systems, showing order-of-magnitude or greater speedups for large chemical kinetic models over a CPU-based code on a quad-core Intel i7 930 processor which used standard `CHEMKIN` [64] and LAPACK [65] libraries for the same operations; it was not clear how/if the CPU code was parallelized. Niemeyer et al. [49] implemented an explicit fourth-order Runge–Kutta integrator for a Tesla C2075 GPU, and found a speedup of nearly two orders of magnitude with a nonstiff hydrogen model when compared with a sequential CPU-code utilizing a single core of an Intel Xeon 2.66 GHz CPU. In a related work, Shi et al. [50] developed a GPU-based stabilized explicit solver on a Tesla C2050 and paired it with a CPU-based implicit solver using a single-core of a quad-core Intel i7 930 that handled integration of the most-stiff chemistry cells in a three-dimensional premixed diesel engine simulation; the hybrid solver was 11–46 $\times$ faster than the implicit CPU solver. Le et al. [66] implemented GPU versions of two high-order shock-capturing reactive-flow codes on a Tesla C2070, and found a 30–50$\times$ speedup over the baseline CPU version running on a single core of a Intel Xeon X5650. Stone and Davis [51] implemented the implicit VODE [42] solver on a Fermi M2050 GPU and achieved an order of magnitude speedup over the baseline CPU version running on a single core of a AMD Opteron 6134 Magny-Cours. They also showed that GPU-based VODE exhibits significant thread divergence, as expected due to its complicated program flow compared with an explicit integration scheme. Furthermore, Stone and Davis [51] found that a per-thread implementation outperforms a per-block version of the same algorithm for $\sim 10^4$ independent IVPs or more; the per-block implementation reached its maximum speedup for a smaller number of IVPs ($\sim 10^3$). Niemeyer and Sung [52] demonstrated an order-of-magnitude speedup for a GPU implementation of a stabilized explicit second-order Runge–Kutta–Chebyshev algorithm on a Tesla C2075 over a CPU implementation of VODE on a six-core Intel X5650 for moderately stiff chemical kinetics. They also investigated levels of thread divergence due to differing integrator time-step sizes, and found that it negatively impacts overall performance for dissimilar IVP initial conditions in a thread-block. Sewerin and Rigopoulos [53] implemented a three-stage/fifth-order implicit Runge–Kutta GPU method [67] on a per-block basis for high-end (Nvidia Quadro 6000) and consumer-grade (Nvidia Quadro 600) GPUs, as compared to a standard CPU (two-core, four-thread Intel i5-520M) and a scientific workstation (eight-core, 16-thread Intel Xeon E5-2687W) utilizing a message passing interface for parallelization; the high-end GPU was at best 1.8 $\times$ slower than the workstation CPU (16 threads), while the consumer level GPU was at best 5.5 $\times$ slower than the corresponding standard CPU (four threads). Yonkee and Sutherland [68] implemented accelerated evaluations of thermodynamic parameters, multicomponent transport

properties, and species production rates on both the CPU and GPU, achieving speedups over serial evaluation between 8–13 × on a 16-core CPU and 20–40 × on the GPU. In addition, ~9 × and ~25 × speedups were achieved for the simulation of a partially premixed methanol flame for solving partial differential equations (PDEs) on 16 CPU cores and the GPU, respectively.

While increasing numbers of studies have explored GPU-based chemical kinetics integration, there remains a clear need to find or develop integration algorithms simultaneously suited for the SIMT parallelism of GPUs (along with similar accelerators) and capable of handling stiffness. In this work we will investigate GPU implementations of several semi-implicit and implicit integration techniques, as compared with their CPU counterparts and the baseline CPU `CVODE` [43] algorithm. Semi-implicit methods do not require solution of non-linear systems via Newton iteration—as required for implicit integration techniques—and instead solve sequences of linear systems [67]; accordingly these techniques are potentially better suited for SIMT acceleration due to an expected reduction of thread divergence (for a per-thread implementation) compared with implicit methods.

Several groups [69, 70] previously suggested so-called matrix-free methods as potential improvements to the expensive linear-system solver required in standard implicit methods. These methods do not require direct factorization of the Jacobian, but instead use an iterative process to approximate the action of the factorized Jacobian on a vector. Furthermore, Hochbruck and Lubich [71, 72] demonstrated that the action of the matrix exponential on a vector obtained using Krylov subspace approximation converges faster than corresponding Krylov methods for the solution of linear equations. Others explored these semi-implicit exponential methods for applications in stiff chemical systems [73, 74] and found them stable for time-step sizes greatly exceeding the typical stability bounds.

Since GPU-based semi-implicit exponential methods have not been demonstrated in the literature, we aim to conduct a systematic investigation to test and compare their performance to other common integration techniques. Finally, we will study the three-stage/fifth-order implicit Runge–Kutta algorithm [67] investigated by Sewerin and Rigopoulos [53] here to determine the impact of increasing stiffness on the algorithm and the performance benefits of using an analytical Jacobian matrix, such as that developed by Niemeyer et al. [44, 75, 76].

Recently, implicit methods improved using adaptive preconditioners have shown promise in reducing integration costs for large kinetic models, compared with implicit methods based on direct, dense linear algebra [77]. These require use of linear iterative methods in addition to the standard Newton iteration, and thus we expect increased levels of thread-divergence (and integrator performance penalties) for the per-thread approach used in this work. However, this area merits future study.

The rest of the chapter is structured as follows. Section 2.2 lays out the methods and implementation details of the algorithms used here. Subsequently, Section 2.3 presents and discusses the performance of the algorithms run using a database of partially stirred reactor thermochemical states, with particular focus on the effects of thread divergence and memory traffic. Further, this work is a starting point to reduce the cost of reactive-flow simulations with realistic chemistry via SIMT-accelerated chemical kinetics evaluation. Thus, we explore the potential impact of current state-of-the-art GPU-accelerated stiff chemical kinetic evaluation on large-scale reactive-flow simulations in Section 2.3, while identifying the most promising future directions for GPU/SIMT accelerated chemical kinetic integration in Section 2.4. The source code used in this work is freely available [78]. Appendix B discusses the validation and performance data, plotting scripts, and figures used in creation of this chapter, as well as unscaled plots of integrator runtimes and characterizations of the partially stirred reactor data for this work.

## 2.2 Methodology

In this section, we discuss details of the algorithms implemented for the GPU along with third-party software used. The generation of testing conditions will be discussed briefly, and the developed solvers will be verified for expected order of error.

### 2.2.1 Integration techniques

| Method | CPU | GPU |
|---|---|---|
| `CVODE` | × | |
| `Radau-IIA` | × | × |
| `exp4` | × | × |
| `exprb43` | × | × |

Table 2.1: The solvers used in this study, and platforms considered for each.

We investigated GPU implementations of three integration methods in this work, namely `RadauIIA` [67], `exp4` [72], and `exprb43` [79], comparing them against equivalent CPU versions and a CPU-only implicit algorithm `CVODE` [15, 43]. Table 2.1 lists these solvers and their corresponding platforms. While we describe important details or changes made in this work, full descriptions of all algorithms may be found in the cited sources. The `pyJac` software [44, 75, 76] provided subroutines for both chemical source terms and the analytical constant-pressure, mass-fraction-based Jacobian matrix used by CPU- and GPU-based algorithms. We evaluated the relative performance impact of using a finite-difference Jacobian matrix (as compared with an analytical Jacobian) for both platforms with a first-order finite difference method based on that

of CVODE [43]. pyJac also provided the chemical source terms used by the finite-difference Jacobian in all cases. We direct readers to our previous work [44, 76] for verification and performance assessments of pyJac itself.

First, the CVODE solver [15, 43] (part of the SUNDIALS suite [80]) provided the baseline performance of a typical CPU-based (maximum of fifth-order) implicit integration technique. In addition, we developed CPU versions of the methods under investigation for direct comparison to the high-order implicit technique. These include the three-stage/fifth-order implicit Runge–Kutta algorithm [67] (Radau-IIA), the fourth-order exponential Rosenbrock-like method of Hochbruck et al. [72] (exp4), and the newer fourth-order exponential Rosenbrock method [79] (exprb43). For the exponential methods, we used the method of rational approximants [81] paired with the Carathédothy–Fejér method [82, 83] to approximate the action of the matrix exponential on a vector, as suggested by Bisetti [73]. This technique relied on the external FFTW3 library [84, 85]. However, unlike the approach of Bisetti [73], we developed a custom routine based on the algorithm presented by Stewart [86] to perform LU decomposition of the Hessenberg matrix resulting from the Arnoldi iteration. Convergence of the Arnoldi iteration algorithm was computed using the second term of the exponential matrix/vector product infinite series, as suggested in several works [73, 87]. The exponential integrators used a rational approximant of type $(10, 10)$ as suggested by Bisetti [73]. To ensure high performance of CPU-based methods, the Intel MKL library version 11.3.2 handled linear algebra (i.e., BLAS/LAPACK) operations.

Next, we developed GPU versions of the Radau-IIA, exp4, and exprb43 methods. These follow the same descriptions as the CPU versions, but require specialized implementations of several BLAS and LAPACK methods, mostly related to LU factorization of the Jacobian or Hessenberg matrices. All GPU routines were developed using the NVIDIA CUDA framework [58, 59], and a block-size of 64 threads (8 blocks per SM) was found to be most efficient for all solvers. All solvers used adaptive time-stepping techniques; the Radau-IIA and CVODE integrators have built-in adaptive time-stepping, while the exponential methods, exp4 and exprb43, used a standard adaptive time-stepping technique [67]. The adaptive time stepping procedures of all integrators used absolute and relative tolerances of $10^{-10}$ and $10^{-6}$, respectively, throughout the work. Finally, the Jacobian was reused on a per-thread (per-IVP) basis according to the built-in rules for the implicit methods, and only recomputed on step failures for the exponential methods.

### 2.2.2 Testing conditions

In order to measure the performance of the integrators for realistic conditions, a database of thermochemical states covering a wide range of temperatures and species mass fractions was generated using a previously developed constant-pressure stochastic partially stirred reactor

(PaSR) code [44, 75]; the details of the PaSR implementation may be found in Appendix A. We selected two chemical kinetic models to span the range of model sizes typically used in high-fidelity simulations: the hydrogen model of Burke et al. [88] with 13 species and 27 reactions, and the GRI-Mech 3.0 model for methane with 53 species and 325 reactions [89]. The PaSR simulations were performed at the conditions listed in Table 2.2 for 10 residence times to reach a statistical steady state; Niemeyer et al. [44] describe the PaSR simulation process in greater detail, which follows approaches used by others [37, 38, 90]. The PaSR particles were initialized using the equilibrium state, and gradually move away from equilibrium conditions due to mixing, inflow, and outflow. In order to reduce the influence of equilibrium conditions on the solution runtime trends for small numbers of IVPs, the first 1000 datapoints were removed from each database; this corresponds to a single pairing time, $\tau_{\text{pair}}$, the time interval at which selected particles in the reactor are randomly swapped with inflowing particles. At this point in the simulation, $\sim 80\,\%$ of the particles were at or near an equilibrium state, and by the 5000th datapoint only $\sim 20\,\%$ of the particles were near equilibrium. The hydrogen and GRI-Mech 3.0 databases consisted of 899,900 and 449,900 total conditions, respectively. Further characterization of the PaSR conditions used in this work can be found in Appendix B and our previous study [44].

| Parameter | $H_2$/air | $CH_4$/air |
|---|---|---|
| $\phi$ | 1.0 | |
| $T_{\text{in}}$ | 400, 600, and 800 K | |
| $p$ | 1, 10, and 25 atm | |
| $N_p$ | 100 | |
| $\tau_{\text{res}}$ | 10 ms | 5 ms |
| $\tau_{\text{mix}}$ | 1 ms | 1 ms |
| $\tau_{\text{pair}}$ | 1 ms | 1 ms |

Table 2.2: PaSR parameters used for hydrogen/air and methane/air premixed combustion cases, where $\phi$ indicates equivalence ratio, $T_{\text{in}}$ is the temperature of the inflowing particles, $p$ is the pressure, $N_p$ is the number of particles in the reactor, $\tau_{\text{res}}$ is the residence time, $\tau_{\text{mix}}$ is the mixing time, and $\tau_{\text{pair}}$ is the pairing time.

### 2.2.3 Solver verification

To investigate the correctness of the developed solvers, the first 10,000 conditions in the hydrogen database were integrated by each solver using a global time-step size of $10^{-6}$ s. The error for condition $i$ was then determined using the weighted root-mean-square error

$$E_i(t) = \left\| \frac{y_i(t) - \hat{y}_i(t)}{\text{atol} + \hat{y}_i(t) \times \text{rtol}} \right\|_2 , \tag{2.1}$$

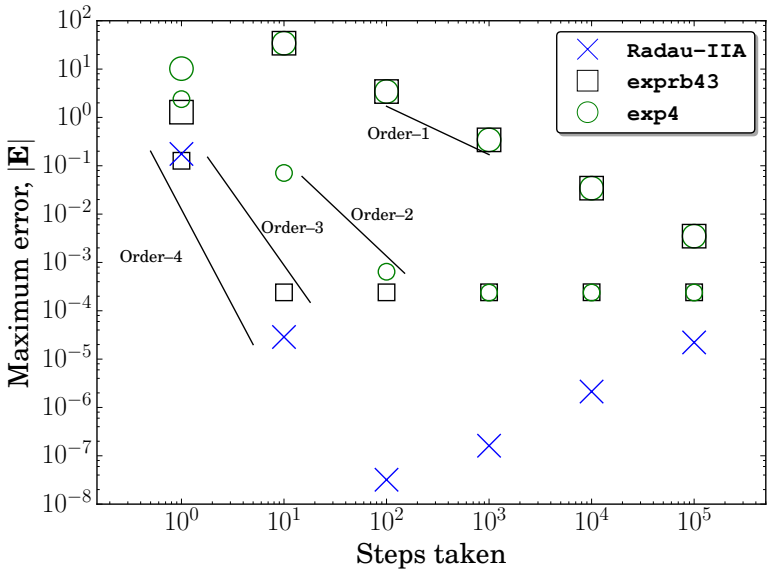



Figure 2.2: Maximum error of the three CPU solvers as a function of the total number of internal integration steps taken (corresponding to decreasing time-step size). Larger square and circle symbols indicate the use of Krylov subspace approximations with the exponential methods, while the smaller symbols indicate the use of "exact" Krylov subspaces. Data, plotting scripts, and figure file are available under CC-BY [60].

where the $y_i(t)$ is the solution obtained from the various solvers, atol/rtol are the absolute/relative tolerances, and $\hat{y}_i(t)$ is the "true" solution obtained via `CVODE` using the same global time-step of $\Delta t = 10^{-6}$ s and absolute/relative tolerances of $10^{-20}$ and $10^{-15}$, respectively; note that the more stringent tolerances were used only to obtain the "true" solution. The maximum error over all conditions:

$$|\mathbf{E}| = \max_{i=1,\ldots,10,000} \{E_i(t)\} \tag{2.2}$$

was then used to measure the error of each solver. The error measurement used the same tolerances as for the performance testing (atol $= 10^{-10}$ and rtol $= 10^{-6}$, respectively). The constant internal time-step size was then varied from $10^{-6}$–$10^{-11}$ s—corresponding to $10^0$–$10^5$ internal integration steps—to measure the convergence rates of the three solvers used in this study.

Figure 2.2 shows the convergence of error for the CPU solvers with decreasing internal time-step size, shown as increasing number of integration steps taken. The error of the `Radau-IIA` integrator drops nearly four orders of magnitude when changing from a single internal time step of $10^{-6}$ s to ten internal time steps of $10^{-7}$ s each, i.e., fourth-order convergence. Increasing the number of integration steps—by further reducing the internal time-step size—past this point results in one further drop in error (of order $\sim 3$); however for more than $10^3$ steps the overall error begins to climb due to accumulation of local error. Since the `Radau-IIA` solver is nominally

fifth-order, it is unclear whether we are observing order reduction due to the stiffness of the problem, use of a numerically obtained "true" solution, or an accumulation of local error. Although a more accurate assessment of convergence order might be achieved through use of a stiff sample problem with an analytical solution—e.g., HIRES [67] or ROBER [67, 91]—direct validation with the problem at hand was conducted here.

The exponential solvers utilizing an approximate Krylov subspace exhibit larger levels of error in general, with $|\mathbf{E}| \sim \mathcal{O}(1)$–$\mathcal{O}(10)$ for a single internal integration step of $\delta t = 10^{-6}$ s. As the time-step size is decreased, the convergence of the Arnoldi algorithm is affected by the internal integration time-step size (the matrix exponentials and error estimates are scaled by the internal time-step). To study the effect of the Arnoldi algorithm on error, Fig. 2.2 also presents the error convergence of the exponential integrators with the Krylov approximation error reduced far below the error of the overall method (for larger internal time-steps). Practically, this was accomplished by detecting when the $n$th Krylov subspace vector approaches zero, a condition known as the "happy breakdown" in literature [92]. At this limit, the approximate exponential matrix/vector product approaches the exact value and thus the Krylov approximation error is relatively small compared to the error of the overall method. It is clear that the error induced by the "exact" Krylov subspace is non-zero however, as both methods reach a minimum error around $10^2$ steps and are unaffected by further step-size decreases, in contrast to the `Radau-IIA` solver which exhibits increasing error past this point due to local error accumulation. Figure 2.2 shows that the exponential methods achieve only first-order convergence to the true solution with the approximate Krylov subspace, but both methods converge at higher rates with the "exact" Krylov subspace. The nominal fourth-order convergence of the `exp4` algorithm is a classical nonstiff order, and thus order reduction is expected for stiff problems [73, 93]; the `exp4` solver reaches roughly second-order convergence with the "exact" Krylov subspace. The `exprb43` solver reaches third-order convergence with the "exact" Krylov subspace. Similar to the discussion on the `Radau-IIA` convergence order, it is difficult to determine whether order reduction has occurred due to problem stiffness, the use of a numerically obtained "true" solution, or some combination thereof. Furthermore, the error of Krylov subspace approximation dominates the error measurement $|\mathbf{E}|$. From Fig. 2.2 we conclude that all three solvers produce reasonably accurate solutions as compared with `CVODE`. Additionally, although not shown, the GPU solvers produce identical results.

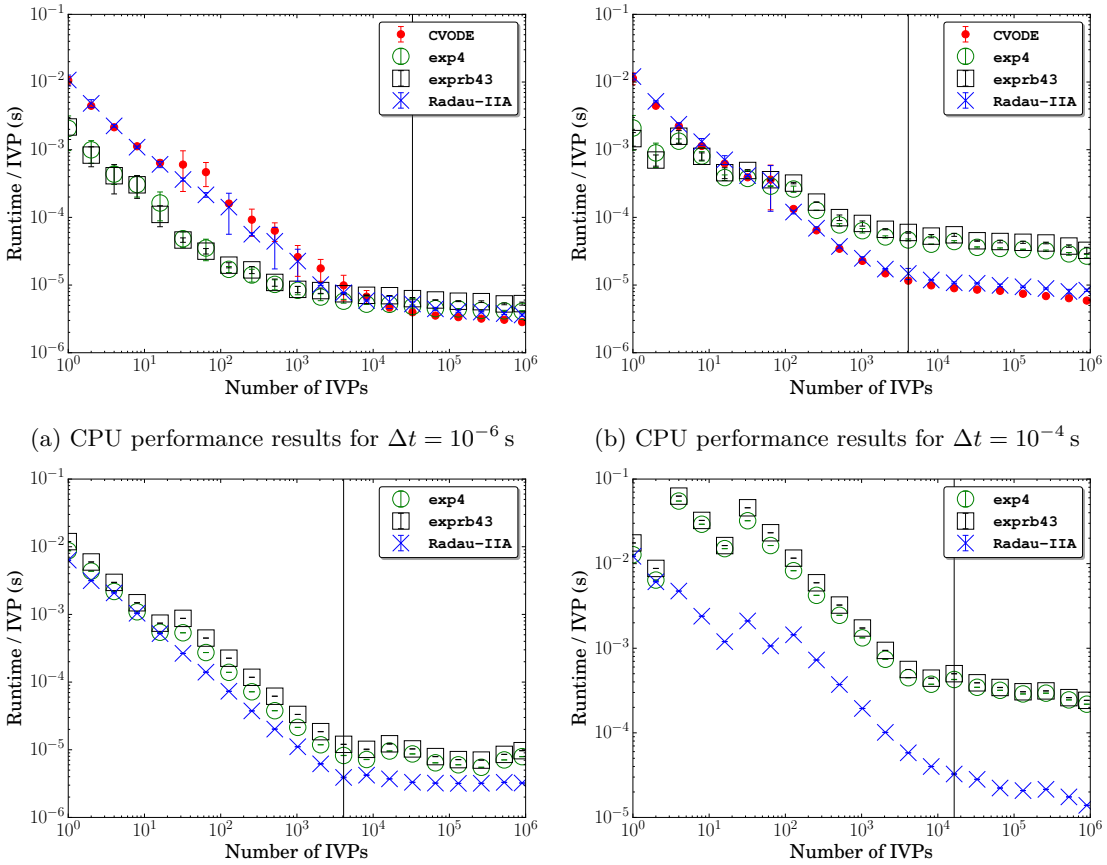
## 2.3 Results and discussion

We studied the performance of the three integrators by performing constant-pressure, homogeneous reactor simulations with two global integration time-step sizes, $\Delta t = 10^{-6}$ s and $\Delta t = 10^{-4}$ s, for each integrator. Initial conditions were taken from the PaSR databases described in Section 2.2.2. A larger global time step induces additional stiffness and allows evaluation of the performance of the developed solvers on the same chemical kinetic model with varying levels of stiffness. In reactive-flow simulations, the chemical integration time-step is typically determined by the flow time-scale and stability requirements determined by the Courant–Friedrichs–Lewy number. Typical global time-step values of reactive-flow simulations are not always clear in the literature, as adaptive time-stepping is often used, or the global time-step size is simply not reported; our own experience suggests global time-step sizes ranging from $10^{-7}$ s to $10^{-4}$ s. The global time-step size used in a given simulation depends highly on the problem and numerical methods, but large-eddy simulations usually require higher time resolution than Reynolds-averaged Navier–Stokes simulations [94]. Hence, the global time-step sizes we selected for study represent realistic values used in large-eddy [95, 96] and Reynolds-averaged Navier–Stokes [97, 98] simulations.

Runtimes are reported as the average over five runs, where each run started from the same set of PaSR conditions. All CPU integrators were compiled using `gcc 4.8.5` (with the compiler options "`-O3 -funroll-loops -mtune=native`") and executed in parallel via OpenMP on four ten-core 2.2 GHz Intel Xeon E5-4640 v2 CPUs with 20 MB of L3 cache memory, installed on an Ace Powerworks PW8027R-TRF+ with a Supermicro X9QR7-TF+/X9QRi-F+ baseboard. OpenMP was used to parallelize on a per-condition basis; i.e., each individual OpenMP thread was responsible for integrating a single chemical kinetic IVP, rather than cooperating with other OpenMP threads to solve the same. A six-core 2.67 GHz Intel Xeon X5650 CPU hosted the GPU integrators, which were compiled using `nvcc 7.5.17` (with compiler options "`-arch=sm_20 -O3 -maxrregcount 63 --ftz=false --prec-div=true --prec-sqrt=true --fmad=false`") and run on a single NVIDIA Tesla C2075 with 6 GB of global memory. Reported runtimes for the GPU-based algorithms include time needed for CPU–GPU data transfer before and after each global time step; in addition, the function `cudaSetDevice()` initialized the GPU before timing to avoid any device initialization delay. The open-source `pyJac` software [44, 75, 76] produced CPU and GPU custom source-code functions for the chemical source terms and analytical Jacobian matrix evaluation. Finally, the L1/shared-memory cache was set to prefer a larger L1 cache using the `cudaDeviceSetCacheConfig()` function.

### 2.3.1 Runtime performance

For all cases in this section, the integrator runtimes are presented as the runtime per IVP solved, for two reasons. First, saturation of the available computational resources becomes visually apparent (transition from a nearly linear decrease to a flat trend), and second, it allows certain other performance trends (e.g., the effects of thread divergence) to be easily highlighted. The presentation of the performance data in raw form is also available in the supplementary material for completeness.




(a) CPU performance results for $\Delta t = 10^{-6}$ s

(b) CPU performance results for $\Delta t = 10^{-4}$ s

(c) GPU performance results for $\Delta t = 10^{-6}$ s

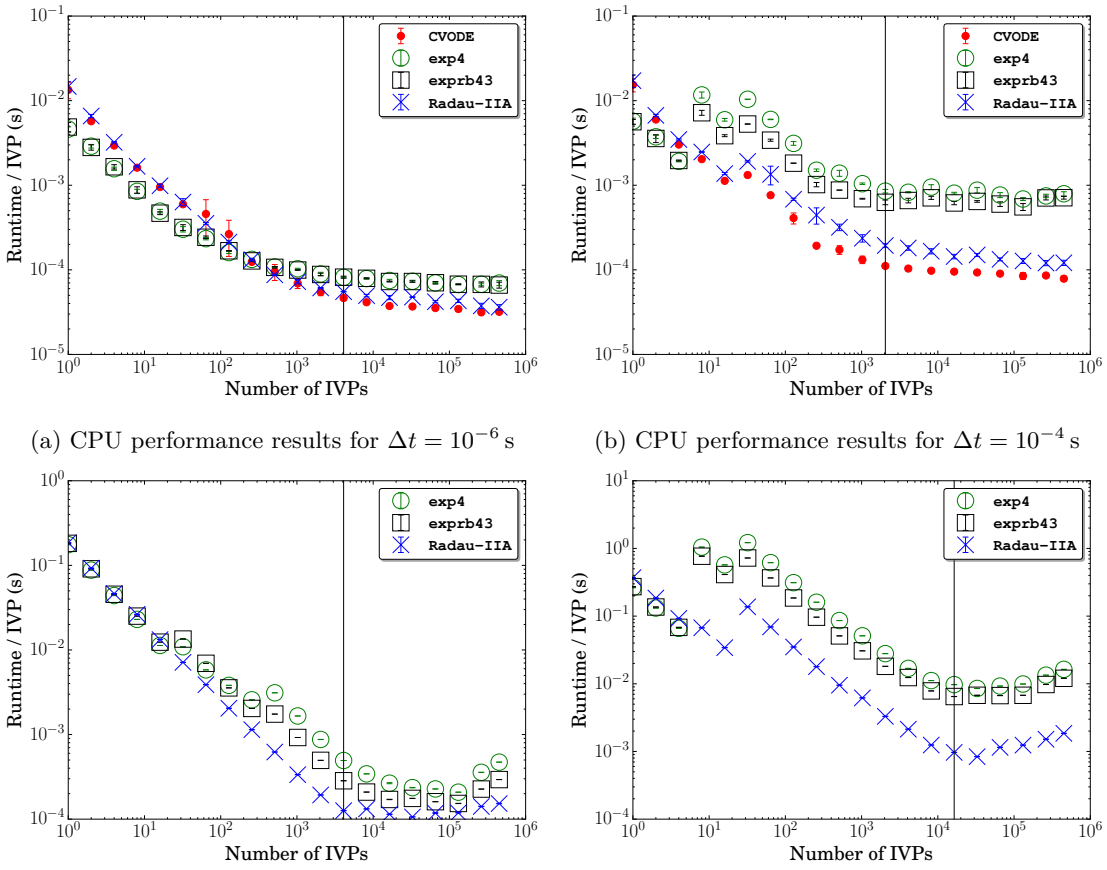
(d) GPU performance results for $\Delta t = 10^{-4}$ s

Figure 2.3: Average runtimes of the integrators on the CPU and GPU, scaled by the number of IVPs, for the hydrogen model at two different global time-step sizes. Estimation of where the runtime per IVP levels off to a constant value (based on the results for `CVODE`/`Radau-IIA` for the CPU/GPU, respectively) is marked with a vertical line for all cases. Error bars indicate standard deviation. Data, plotting scripts, and figure files are available under CC-BY [60].

Figure 2.3 shows the runtimes of the CPU and GPU integrators for the hydrogen model. In Fig. 2.3a the runtimes per IVP for the CPU integrators for a single global time-step of $\Delta t = 10^{-6}$ s decrease approximately linearly with the number of IVPs for small numbers of initial conditions (shown here on a log-log plot). For small numbers of IVPs, the exponential integrators are faster than the implicit integration techniques due to the modest stiffness of the hydrogen model; even with many near-equilibrium states removed from the beginning of the PaSR

database, the model is not particularly stiff for this small time-step size. Larger numbers of IVPs begin to saturate the CPU resources, and the runtime per IVP levels off to a more constant value; vertical lines are shown in Fig. 2.3 where the relative change in runtime per IVP between successive data-points is first smaller than 15 % (based on the results for `CVODE`/`Radau-IIA` for the CPU/GPU respectively). Eventually, relatively more stiff conditions are encountered and the performance of the implicit integration techniques catches up and then surpasses that of the exponential integrators; `CVODE` is the most efficient solver on the CPU when solving more than $10^4$ IVPs; however, `CVODE` is only $\sim$1.87$\times$ faster than the slowest solver (`exprb43`) on the whole database. Figure 2.3c shows the performances of the GPU integrators for the smaller global time-step size, which exhibit similar trends as the CPU solvers: a linearly decreasing solution cost that reaches a roughly constant value beyond $10^3$–$10^4$ IVPs. Unlike for the CPU solvers, the GPU-based `Radau-IIA` performs faster than the exponential solvers for all numbers of IVPs. As will be seen in Section 2.3.3, both solver classes experience minimal thread divergence due to differing internal integration time-step sizes in this case. Therefore, we conclude that the relatively slower runtimes per IVP for the exponential algorithms on the GPU results from thread divergence in the Arnoldi iteration—caused by varying Krylov subspace sizes between threads. Figures 2.3b and 2.3d show the performance of the integration algorithms on both platforms for the hydrogen model with a single larger global time step ($\Delta t = 10^{-4}$ s). The performances of the CPU integration algorithms show similar trends to those of the smaller global time-step size case: decreasing cost per IVP before reaching a more constant performance for higher numbers of IVPs. The larger global time-step size induces additional stiffness, and the implicit solvers are more efficient for most numbers of IVPs; `CVODE` is again the most efficient CPU solver. Figure 2.3d shows the performance of the GPU solvers for the larger global time-step size. The exponential solvers exhibit significant spikes in computational cost when changing from 2–4 and 16–32 IVPs, with the latter mimicked somewhat by the implicit `Radau-IIA` solver. A jump in solution cost between 2–4 IVPs is also present for the CPU exponential integrators, indicating stiffness as the primary cause. On the other hand, between 16–32 IVPs the CPU exponential solvers exhibit only a very minor performance decrease, while the GPU-based `Radau-IIA` also shows a decrease in performance at the same point—a trend completely absent in the CPU `Radau-IIA` version. These factors indicate that thread divergence also plays a key role in the performance trend here, and will be investigated further in Section 2.3.3. As in case of the smaller global time-step size, the `Radua-IIA` solver is the most efficient GPU algorithm in all cases.

Figure 2.4 shows the runtime of the integrators for the GRI-Mech 3.0 model. Similar to the hydrogen case for the smaller global time-step size, the CPU exponential integrators are more efficient (Fig. 2.4a) for the near-equilibrium conditions at the beginning of the database. For




(a) CPU performance results for $\Delta t = 10^{-6}$ s

(b) CPU performance results for $\Delta t = 10^{-4}$ s

(c) GPU performance results for $\Delta t = 10^{-6}$ s

(d) GPU performance results for $\Delta t = 10^{-4}$ s

Figure 2.4: Average runtimes of the integrators, scaled by number of IVPs, on the CPU and GPU for the GRI-Mech 3.0 model at two different global time-step sizes. Estimation of where the runtime per IVP levels off to a constant value (based on the results for `CVODE`/`Radau-IIA` for the CPU/GPU respectively) is marked with a vertical line for all cases. Error bars indicate standard deviation. Data, plotting scripts, and figure files are available under CC-BY [60].

larger numbers of conditions, the implicit integrators are more efficient, and `CVODE` again performs the fastest. Compared with the hydrogen model (Fig. 2.3a), the `CVODE` performs better than the exponential algorithms for the GRI-Mech 3.0 model with the small global time-step size (Fig. 2.4a), reaching a speedup of 2.18 × over `exp4` on the whole database; this results from the higher stiffness present in the model. This performance gap between the CPU implicit/exponential integrators increases for the larger global time-step size (Fig. 2.4b); `CVODE` is 10.1 × faster than `exp4` on the whole database. Comparing the performance of the CPU implicit solvers between the two chemical kinetic models shows roughly an order-of-magnitude performance decrease for both global time-step sizes. This phenomena, due largely to the increase in model size, is also seen for the `Radau-IIA` GPU solver for the smaller global time-step size; the performance of which decreases by just over an order of magnitude. However, for the larger global time-step size, the GPU-based `Radau-IIA` solver performs roughly two orders-of-magnitude slower compared with the hydrogen case. As will be examined in Section 2.3.3, this dramatic decrease likely results from increased thread divergence in the `Radau-IIA` solver, as well as the increased memory traffic inherent in the larger model. Unlike for the hydrogen model, the `exprb43` solver outperforms `exp4` with the GRI-Mech 3.0 model in almost all cases for the larger global time-step size for both the CPU and GPU. Although the `exprb43` and `exp4` algorithms each require three exponential matrix function approximations per step, a single internal time step of `exprb43` is more expensive due to the extra chemical source term evaluations, matrix multiplications, and higher-order exponential matrix function requirement. As such, the relatively simpler CPU `exp4` integrator outperforms the CPU `exprb43` integrator for the hydrogen model where there is relatively less stiffness. However, as previously discussed the `exp4` algorithm may experience order reduction for stiff problems, and the `exprb43` algorithm typically outperforms `exp4` on both the CPU and GPU in the larger global time-step GRI-Mech 3.0 case as a result.

### 2.3.2 CPU/GPU performance comparison

Comparing the performance of CPU- and GPU-based integrators in a meaningful way is challenging. First, the vastly different nature of the processing cores in each platform eliminates the possibility of comparing performance normalized by core count. In addition, the floating-point operation count is not readily available for chemical kinetics integration—unlike many GPU-accelerated applications where the number of operations required to solve the problem is known, e.g., as in linear-algebra operations or fast Fourier transforms—which precludes comparing performance on the basis of floating-point operations per second (FLOPS). Although the runtimes of the GPU integration algorithms can be directly compared with that of

the CPU-based solvers (and often are), these figures do not provide much useful information. For instance, if a GPU algorithm performs 10 × faster than its equivalent on two six-core CPUs, how does this compare to two eight-core CPUs, etc.?

For researchers in numerical combustion, two issues stand out as particularly important for performance evaluation: runtime and cost. As established in Section 2.1, large-scale reactive-flow simulations with realistic chemical kinetic models are extremely computationally expensive, and remain outside of the capabilities of most in the field. With this in mind, we ask, for a given simulation, what is the effect on the overall runtime of adding more CPU cores compared with adding GPU accelerators? In addition, if a budget is allocated to expand available computational resources, how might these funds be best allocated? To answer these questions, we derived an estimate of the number of CPU cores required for equivalent performance on the GPU.

A nominal performance metric for both the CPU- and GPU-based integration algorithms must first be obtained. As the most efficient solvers in all cases with large numbers of IVPs are `CVODE` for the CPU and `Radau-IIA` for the GPU, these algorithms will be considered the performance benchmarks. Furthermore, most large-scale simulations consist of millions of cells (or more), and therefore we only consider the performance limit of each algorithm (i.e., the cost per IVP of each algorithm in the region where this cost reaches an approximately constant value). To this end, the previously discussed threshold—the first relative change in runtime per IVP between successive data-points smaller than 15 % (based on `CVODE`/`Radau-IIA` for the CPU/GPU accordingly)—is used, and marked as vertical lines on Figs. 2.3 and 2.4. The cost per IVP above and including these thresholds was averaged and forms our nominal performance measure. The CPU performance measure must also be normalized by the total number of cores used: 40.

Table 2.3 presents the ratios of these performance measures, which give estimates for the number of CPU cores required to equal the GPU performance for the cases studied. The GPU is roughly equivalent to 12 or more CPU cores for all cases except GRI-Mech 3.0 with the larger global time-step size, and equivalent to at most 38 cores for the hydrogen case with the smaller global time-step size. With the increasing size of the chemical kinetic model, the equivalent CPU core count of the GPU `Radau-IIA` solver drops significantly. As will be discussed in Section 2.3.3, this drop in performance is primarily due to higher memory traffic requirements, however increased levels of thread divergence also play a role. Although this work represents the current state-of-the-art for implicit integration of stiff chemical kinetic IVPs on the GPU, it is clear that more effort is required to improve GPU performance for larger chemical kinetic models. Approaches to mitigate these issues will be discussed in the subsequent section.

At the time of writing, the ten-core Intel Xeon E5-4640 v2 CPU used in this study was listed for a recommended customer price of $2725 [99], while a new Tesla C2075 GPU is available for

| Global time-step size | # equivalent CPU cores | |
|---|---|---|
| | Hydrogen | GRI-Mech 3.0 |
| $10^{-6}$ s | 38 | 12 |
| $10^{-4}$ s | 15 | 3 |

Table 2.3: The number of CPU cores (roughly) required for equivalent performance to a single GPU for the combinations of chemical kinetic models and global time-step sizes studied.

∼$1400 [100]. These prices are only rough estimates of the actual cost of these devices, since the actual price for the Intel CPU may be significantly less in a configured server node, while the Tesla C2075 is no longer sold directly by NVIDIA—thus the prices are variable. Furthermore, the performance decrease using an older, cheaper CPU (e.g., the Intel Xeon X5650 used as host processor for the GPU simulations in this work) may not be that large. However, combined with the equivalent core counts in Table 2.3, this information suggests that the Tesla C2075 is a reasonable investment to supplement computing power for chemical-kinetic integration in large-eddy simulations.

### 2.3.3 Effects of thread divergence and memory traffic

Thread divergence and memory traffic are two performance concerns particularly important for chemical kinetics integration on GPU and SIMT platforms. Slowdown due to memory traffic for a GPU integration algorithm implemented on a per-thread basis primarily results from the small amount of on-chip memory available. Implicit integration algorithms, which typically require storage of the Jacobian matrix and/or factorized forms thereof, can quickly overwhelm the registers and L1 cache memory available to each thread and cause many slow global memory accesses. Reformulating the chemical kinetic equations to generate sparse Jacobian matrices [101] would greatly benefit GPU-based integration algorithms due to the reduced memory requirements, and in addition enable use of sparse multiplication/factorization algorithms (from which a CPU-based algorithm would also benefit); this is a planned improvement to the `pyJac` software [44, 75].* Further, the Tesla C2075 GPU used in this study was originally released nearly five years ago and is several generations old; the newer Tesla K40 is available for a similar price, $2950 [102], as the Xeon E5-4640 v2 CPU used in this study, and has 2× registers available per block [59] and 6.4× as many CUDA cores [103] as the Tesla C2075 used. Using a newer GPU model could significantly improve solver performance for larger models by relieving the scarcity of on-chip memory in a per-thread approach. Finally, a per-block approach may be required to efficiently integrate the largest models on the GPU, due to the much higher amount of cache

*Bisetti [73] demonstrated a method to exploit the underlying sparsity of a dense mass-fraction-based constant-pressure Jacobian matrix (used in this study) to accelerate Jacobian-vector multiplications; however, a reformulation is still more attractive as it enables sparse-LU factorization.

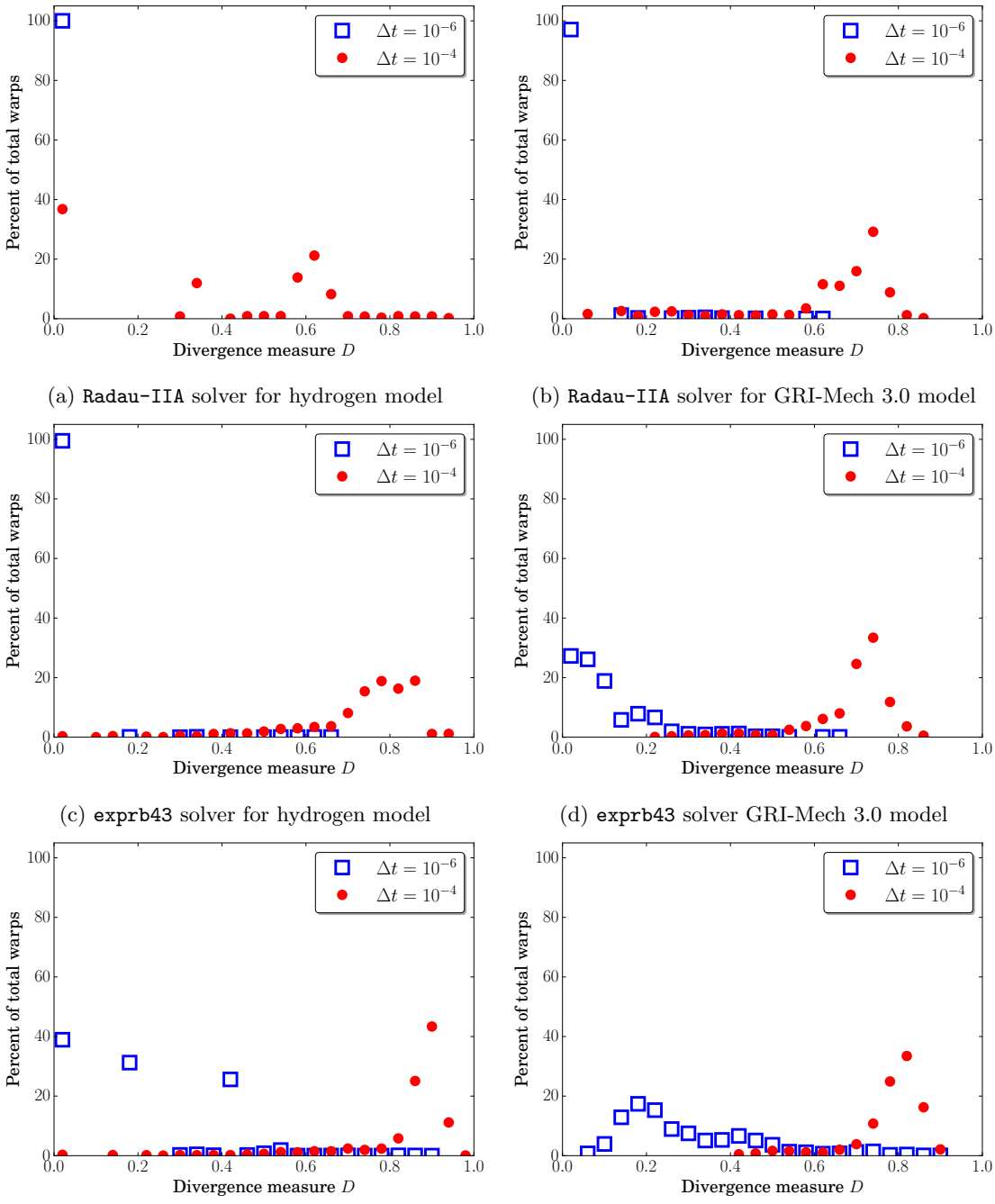
memory allocated for each IVP solution.

The performance penalty due to thread divergence depends both on the cost of the divergent branches as well as the proportion of the warp that executes each branch. For example, if only one thread in a warp executes an expensive branch (e.g., a Jacobian update), the rest of the warp remains idle during that time, and the SM may become severely underutilized. To investigate the effects of thread divergence further, we adopted a modified version of the quantification of thread divergence of Niemeyer and Sung [52]:

$$D = 1 - \frac{\sum_{i=1}^{32} d_i}{32 \times \max\limits_{i=1,\ldots,32} d_i} \,, \tag{2.3}$$

where $d_i$ is the number of internal integrator time steps taken to reach the global time step by thread $i$ in a warp (which consists of 32 threads). $D$ represents the similarity of internal time step counts across threads in a warp—a significant source of thread divergence. If all threads in a warp use identical internal integration time steps and thus the warp experiences no thread divergence from this source, then $D = 0$; however, if a warp experiences an unbalanced number of internal integration time steps, then $D \to 1$. Differing internal time-step sizes are not the only source of thread divergence for the GPU integration algorithms. For instance, threads in a warp may use different Krylov subspace sizes for the exponential integrators or different numbers of Newton iterations for the `Radau-IIA` solver. Indeed, Section 2.3.1 notes that we suspect thread divergence from differing Krylov subspace sizes as the reason the exponential solvers are less efficient for small numbers of IVPs for the hydrogen model with the small global time-step size. However, these operations clearly cost less than an entire internal integration step (in which they are embedded) and thus we look only at the thread divergence of internal integration time steps. Thread divergence of such operations within an internal integration step could play an important role and will be investigated in our future work.

Figures 2.5a and 2.5b show the distribution of the divergence measure $D$ for the `Radau-IIA` solver with both global time-step sizes and kinetic models when run on 262,144 IVPs, spread across 8192 warps. For both kinetic models with the smaller global time-step size, nearly 100 % of the warps had a divergence measure near zero. Increasing the global time-step size causes the number of warps with high levels of thread divergence (e.g. $D > 0.5$) to increase for both models. For the hydrogen model, over 40 % of warps were between $D = 0.55$ and $D = 0.65$, and the approximate equivalent CPU core-count (Table 2.3) dropped by 2.5 × between the small and large global time-step sizes. Further, over 75 % of warps were between $D = 0.6$ and $D = 0.8$ for the GRI-Mech 3.0 model for the larger global time-step size, and subsequently a higher drop in performance of 4 × occurred. This observation motivates future work aimed at developing




(a) `Radau-IIA` solver for hydrogen model

(b) `Radau-IIA` solver for GRI-Mech 3.0 model

(c) `exprb43` solver for hydrogen model

(d) `exprb43` solver GRI-Mech 3.0 model

(e) `exp4` solver hydrogen model

(f) `exp4` solver GRI-Mech 3.0 model

Figure 2.5: Thread divergence estimate for the three solvers for both models and global time-step sizes. Data, plotting scripts, and figure files are available under CC-BY [60].

strategies to reduce thread divergence. Potential solutions include adopting an IVP per-block approach [51], reordering IVPs to increase similarity of stiffness inside a warp, or synchronizing internal time-step sizes between threads in a warp. However, Figs. 2.5a and 2.5b do not explain the drop in equivalent core count between the hydrogen model and the GRI-Mech 3.0 model for the smaller global time-step size. The minimal thread divergence of the `Radau-IIA` solver for both models at the smaller global time-step size suggests that this drop in performance is primarily caused by the increased memory traffic of the larger model, as well potential thread divergence inside the internal integration step; this further motivates development of a sparse version of the `pyJac` [44, 75] software.

Figures 2.5c and 2.5d show the divergence levels of the `exprb43` GPU solver. Similar to the `Radau-IIA` solver, nearly 100 % of warps for the `exprb43` solver have no thread divergence due to differing internal integration step sizes for the hydrogen model. The `exprb43` thread divergence levels increase somewhat for the GRI-Mech 3.0 model with the smaller time-step size; 27 % of warps still had a divergence measure of $D = 0$, but nearly 63 % of the warps had divergence measures between $D = 0.05$ and $D = 0.2$. With the larger time-step size, the `exprb43` solver experiences significantly more thread divergence for both models. The divergence measure distribution is fairly similar to that of the `Radau-IIA` solver for the GRI-Mech 3.0 model, but most warps experience a divergence measure of $D \sim 0.8$ for the hydrogen model (versus $D \sim 0.6$ for the `Radau-IIA` solver). The semi-implicit solvers deal with stiffness less efficiently, and end up using a greater range of internal time-step sizes between conditions of varying stiffness. This results in an increase in thread divergence levels due to differing internal time-step sizes.

The relatively worse stiffness handling of the `exp4` method is also apparent in Figs. 2.5e and 2.5f; in most cases, significantly more thread divergence is seen for `exp4` than for either of the other two solvers. The `exp4` algorithm is the only solver to show significant thread divergence even for the hydrogen model for the smaller global time-step size. Further, the `exp4` algorithm experiences more thread divergence than the `exprb43` for both models at the larger global time-step size.

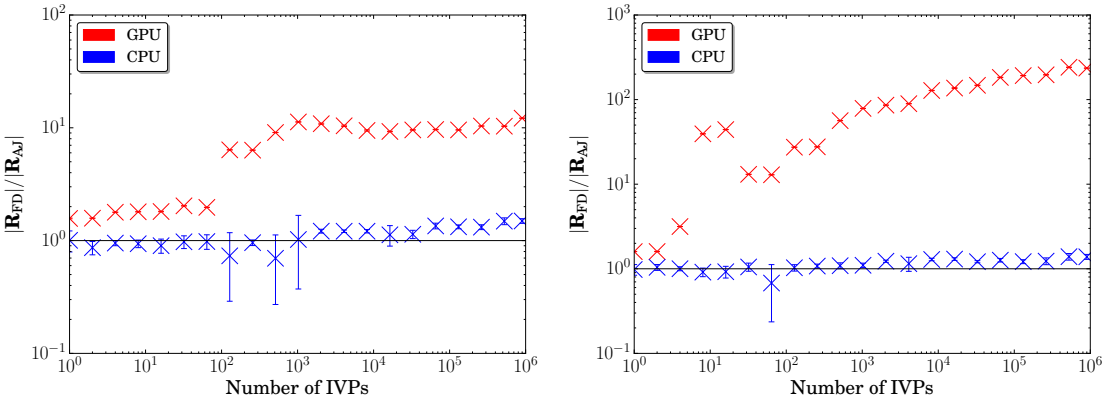
### 2.3.4 Effect of using a finite-difference-based chemical kinetic Jacobian

While it is well established that using an analytical Jacobian matrix can significantly accelerate chemical kinetics integration on the CPU (e.g., [16, 101, 104]), relatively little study has been directed at use of a GPU-based analytical Jacobian. Dijkmans et al. [105] used a GPU-based analytical Jacobian code to accelerate various CPU-based chemical kinetics integration schemes, and our own previous works [44, 76] have detailed the performance of `pyJac`. However, to our knowledge no work using an analytical Jacobian for GPU-based chemical kinetics integration has been published. In this section, we explore the relative performance benefits of the analytical

Jacobian compared with a first-order finite-difference Jacobian on both the CPU and GPU. The exponential methods require an exact Jacobian matrix (rather than an approximation as given by finite-difference methods), so their performance was not considered in this section.

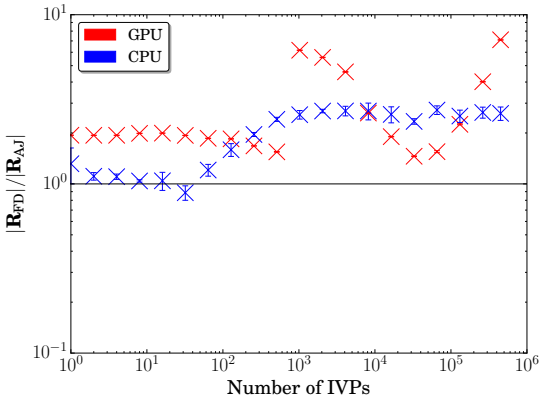
Figure 2.6 shows the speedup achieved on both the CPU and GPU for the `Radau-IIA` algorithm for various cases; the GRI-Mech 3.0 results for the larger global time-step size have been omitted due to long run times. For the hydrogen model (Figs. 2.6a and 2.6b), using the analytical Jacobian offers minimal performance benefit for the CPU-based integrators, reaching a maximum speedup of 1.49 × and 1.39 × for the small and large global time-step sizes, respectively. Our previous work [44] demonstrated that evaluation of the analytical Jacobian was 5.28 × faster on the CPU for the same chemical kinetic model; thus, the minor speedup seen here results from reuse of the Jacobian within the `Radau-IIA` solver, such that integration only requires a few Jacobian evaluations. In some cases the finite-difference Jacobian solver may be faster than the analytical Jacobian solver; although it is difficult to explain the exact cause of this phenomena, differences in the finite-difference Jacobian likely caused the integrator to follow a slightly different instruction path (e.g., with fewer Jacobian updates/chemical source term evaluations) changing the integration cost. However, for large numbers of conditions, the analytical-Jacobian-based CPU solver indeed performs faster than the finite-difference counterpart. In contrast, the analytical-Jacobian-based GPU solver performs significantly faster than the finite-difference GPU solver in all cases for the hydrogen model, reaching a maximum speedup of 12.16 × for the smaller global time-step size. As discussed in Section 2.3.3, significantly higher levels of thread divergence are expected for the larger global time-step size. Correspondingly, the maximum speedup of the GPU solver increases to 240.96 × for the larger global time-step size. Figure 2.6c shows that the speedup of the CPU and GPU solvers reach 2.61 × and 7.11 ×, respectively, for the larger GRI-Mech 3.0 model at the smaller global time-step size.

Upon subsequent investigation of the source of the large speedups that occur when using analytical versus a finite-difference Jacobian evaluation on the GPU, it was revealed that the version of `nvcc` and CUDA toolkit used in this study—`v7.5.17` and `v7.5`, respectively—greatly slows down the finite-difference Jacobian cases when compared with a newer version of the CUDA toolkit (`v8.0.61`). Instead the GPU speedups for the $H_2$/CO model are more similar to those seen on the CPU in Figs. 2.6a and 2.6b. Thus these large speedups are not typical of what should be expected when using analytical Jacobian evaluation on the GPU, the GPU speedup results in Fig. 2.6 are specific to CUDA `v7.5`.




(a) Hydrogen model with $\Delta t = 1 \times 10^{-6}$ s

(b) Hydrogen model with $\Delta t = 1 \times 10^{-4}$ s




(c) GRI-Mech 3.0 model with $\Delta t = 1 \times 10^{-6}$ s

Figure 2.6: Ratio of the average finite-difference Jacobian based integrator runtime $|\mathbf{R}_{\mathrm{FD}}|$ to that of the analytical Jacobian runtime $|\mathbf{R}_{\mathrm{AJ}}|$ for the `Radau-IIA` (CPU/GPU) solvers. Error bars indicate standard deviation, and the horizontal lines show a ratio of one. Data, plotting scripts, and figure files are available under CC-BY [60].

## 2.4 Conclusions

The large size and stiffness of chemical kinetic models for fuels traditionally requires the use of high-order implicit integrators for efficient solutions. Past work showed orders-of-magnitude speedups for solution of nonstiff to moderately stiff chemical kinetic systems using explicit solvers on GPUs [49, 52, 66]. In contrast, work on stiff chemical kinetics integration with implicit GPU solvers has been limited to specialized cases, or failed to surpass current CPU-based techniques. This work demonstrated and compared the performances of CPU- and GPU-based integration methods capable of handling greater stiffness, including an implicit fifth-order Runge–Kutta algorithm and two fourth-order exponential integration algorithms, using chemical source term and analytical Jacobian subroutines provided by the `pyJac` software [44, 75, 76]. By comparing the performance of these algorithms using two chemical kinetic models, including hydrogen with 13 species and 54 reactions [88] and methane with 53 species and 325 reactions [89], and using two global time-step sizes ($10^{-6}$ s and $10^{-4}$ s), we drew the following conclusions:

- For global time-step sizes relevant to large-eddy simulations (e.g., $\Delta t = 10^{-6}$ s), the GPU-based implicit Runge–Kutta method was roughly equivalent to the CPU-based implicit `CVODE` integrator running on 12–38 CPU cores.
- At larger global time-step sizes, the performances of all GPU-based integrators decreased significantly due to thread divergence.
- For a global time-step size relevant to Reynolds-averaged Navier–Stokes simulations (e.g., $\Delta t = 10^{-4}$ s), the GPU-based implicit Runga–Kutta solver performed equivalent to `CVODE` running on 15 cores for the hydrogen model, and just 3 cores for the GRI-Mech 3.0 model.
- The higher memory traffic required due to the size of the GRI-Mech 3.0 model significantly decreased GPU solver performance; a sparse analytical chemical kinetic Jacobian formulation must be developed to achieve high performance for still larger chemical kinetic models on the GPU.
- The exponential solvers were significantly less efficient than the implicit integrators on the CPU and GPU for all relevant cases.

Based on these results, we conclude that the exponential solvers poorly fit the SIMT acceleration paradigm due to high levels of thread divergence combined with the relatively high cost of integration steps due to Arnoldi iteration (as compared with other semi-implicit integration techniques). Instead, we recommend directing further focus on stiff semi-implicit solvers such as (non-exponential) Rosenbrock solvers, explored for the CPU by Stone and Bisetti [104], and inexact Jacobian W-methods [106, 107]. Further improvements to the analytical Jacobian code,

e.g., by using a chemical kinetic system based on species concentrations to increase Jacobian sparsity, are likely to further increase performance of the developed algorithms. Additionally, newer GPUs should be tested to examine the ability of larger cache sizes and more available registers to improve performance by reduction of slow global memory loads/stores; a per-block solution still may need to be adopted for efficient integration of larger chemical kinetic models. However, this work also showed that thread divergence poses a challenge to high performance of GPU-based integration techniques on a per-thread basis. Our future work will therefore include a more comprehensive study of thread divergence, as well as developing methods to mitigate or eliminate its negative performance impact. Finally, new integration techniques will be investigated and paired with work studying the selection of appropriate solvers based on estimated stiffness.

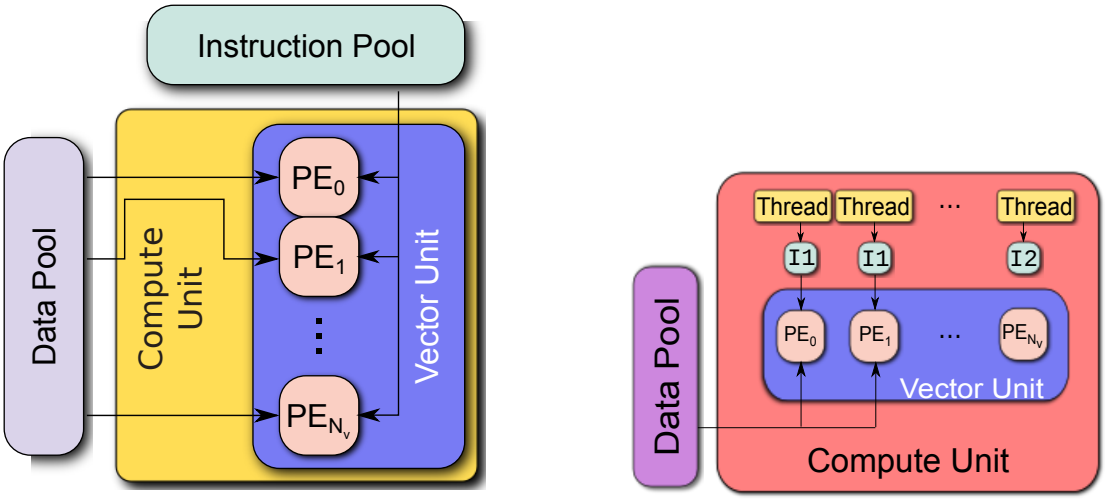
# Chapter 3

# Using SIMD and SIMT vectorization to evaluate sparse chemical kinetic Jacobian matrices and thermochemical source terms

## 3.1 Introduction

Single-Instruction, Multiple-Data (SIMD) and the related Single-Instruction, Multiple-Thread (SIMT) programming are two important vector-processing paradigms used increasingly in scientific computing. The parallel programming standard OpenCL [108] has further enabled adoption of vector processing in scientific computing by providing a common application program interface (API) for execution on heterogeneous systems, e.g., CPU, GPU, or Intel's Many Integrated Core (MIC) architecture. Here we will largely use OpenCL terminology to describe these processing paradigms, as it provides a convenient way to classify otherwise disparate processor types (e.g., CPUs and GPUs). However, the concepts discussed herein broadly apply to SIMD/SIMT processing.

A typical modern CPU contains multiple compute units (i.e., cores), each with specialized vector processing units capable of running SIMD instructions, as Fig. 3.1a depicts. A SIMD instruction uses the vector processor to execute the same floating-point operation (e.g., multiplication, division) on different data concurrently. The vector-width is the number of possible concurrent




(a) Schematic of SIMD processing. A single compute unit (e.g., a CPU core) contains a vector unit with $N_v$ processing elements (PEs), together called a vector-lane. The vector unit executes a single instruction concurrently on multiple data.

(b) Schematic of SIMT processing. A single compute unit (e.g., a GPU streaming multiprocessor) contains many processing elements (PEs) and hosts many threads, each with an instruction to execute (I1, I2). Threads with the same instruction execute concurrently on multiple data while the others must wait (leading to thread divergence).

Figure 3.1: Simple diagrams explaining the fundamentals of the SIMD and SIMT vector-processing paradigms.

operations, typically around two to four in double precision.* Specialized hardware accelerators have also been developed, like Intel's Xeon Phi co-processor (i.e., the MIC architecture), that have tens of cores with wide vector-widths (e.g., 4–8 double-precision operations). Cutting-edge and forthcoming Intel CPUs also include these wide vector-widths, like the Skylake Xeon and Cannon Lake architectures.

Modern GPUs rely on the related computing paradigm of SIMT processing, where a single compute element hosts large numbers of threads (a streaming multiprocessor in Nvidia terminology) [109]. Figure 3.1b depicts a SIMT compute unit, where a group of threads—typically 32, known as a warp on Nvidia GPUs—execute the same SIMT instruction on multiple data concurrently. If some threads must execute a different instruction, they are forced to wait and execute later; this may occur due to if/then branching or predication. This phenomenon, known as thread-divergence, is a key consideration for SIMT processing and can cause serious performance degradation for complicated algorithms [110].

*OpenCL allows for use of vector-widths different from the actual hardware vector-width via implicit conversion, and may provide some performance benefit as Sec. Section 3.3.5 discusses.

### 3.1.1 Related work

Recognizing the need to accelerate chemical-kinetic Jacobian evaluation and factorization, a number of recent works have been published on constructing analytical Jacobian matrices; although as will be discussed at the end of this section, here we offer several key improvements over past efforts. Schwer et al. [101] were among the first to recognize the critical importance of a sparse analytical Jacobian to accelerate chemical kinetic simulations. Later, Safta et al. [111] developed the `TChem` software package, which was one of the first developed that provides analytical Jacobian evaluation. However, `TChem` has several limitations, including incompatibility with modern reaction types—i.e., pressure-dependent Arrhenius (or P-Log) and Chebyshev reactions—and its lack of thread-safety to enable parallel execution [112]. Youssefi [113] explored the importance of analytical Jacobian matrices for time-scale analysis techniques as well as their effect on computational efficiency in zero-dimensional homogeneous reactor simulations.
Bisetti [73] developed an isothermal, isobaric analytical Jacobian code-generation utility; this approach significantly increases Jacobian sparsity, although the chosen isothermal assumption is not typical in most combustion simulations. In the same work Bisetti also provided a novel way to compute dense matrix-vector multiplications resulting from a change of system variables without storing the full Jacobian. Perini et al. [114] developed an analytical Jacobian code for constant-volume combustion, with additional options to increase sparsity (at the expense of strict correctness) and tabulate temperature-dependent properties; they reported an 80 % speedup over a finite-difference-based Jacobian when used in a multidimensional reactive-flow simulation. Gao et al. [115] derived a sparse analytical Jacobian, but did not verify it outside the context of use with an implicit-integration technique. In addition, since the Jacobian was based on an over-constrained system [116], the effect on strict conservation of mass/energy was not studied.
Recently, some groups have developed frameworks for constructing analytical Jacobians for evaluation on modern SIMD or SIMT processors. Dijkmans et al. [105] developed a GPU-based analytical Jacobian code with optional tabulation of temperature-dependent properties, and showed speedups up to 120 × for zero-dimensional chemical kinetic integration with large chemical models (~3000 species). Bauer et al. [117] used warp-specialization to improve GPU-vectorization over a standard data-parallel vectorization approach; they achieved speedups of up to 2.81–3.75 ×, 1.91–2.58 ×, and 1.4–1.5 × for evaluating viscosity, species diffusion, and chemical source terms, respectively. Niemeyer et al. [44] created and verified the open-source analytical chemical kinetic Jacobian code-generator, `pyJac`, which supports parallel execution on CPUs and SIMT execution on GPUs; `pyJac` enables a speedup of 3–7.5 × over a finite-difference Jacobian on the CPU.
Relevant to all of the aforementioned efforts, Hansen and Sutherland [116] explored the choice of

thermochemical state vectors and the resulting effect on consistency and errors in conserved properties such as mass and energy. They also characterized how the choice of state vector affects implicit/linearly implicit integration algorithms and chemical mode analysis techniques. Overall they found that while many literature Jacobian formulations are not strictly correct or over-specified, such flaws negligibly affect Newton–Krylov methods—perhaps because the incorrect Jacobian reasonably approximates the true Jacobian. On the other hand, linearly implicit algorithms like Rosenbrock methods and analysis techniques like chemical explosive mode analysis [118] need accurate and correct Jacobians.

A number of recent works have investigated using high-performance SIMT devices like GPUs to accelerate reactive-flow and chemical kinetics simulations, as reviewed in Chapter 2. In contrast, SIMD-based chemical kinetics evaluation/integration have been studied far less. Linford et al. [119] implemented a three-stage, second-order Rosenbrock integrator for atmospheric chemical kinetics on the CPU, GPU, and cell broadband engine (CBE)—a specially designed vector processor—and found speedups regularly exceeding 25 × over a serial CPU implementation. Kroshko and Spiteri [55] implemented a SIMD-vectorized third-order stiff Rosenbrock integrator for atmospheric chemistry on the CBE and found a speedup of 1.89 × (a parallel scaling efficiency of 94 %) over a serial version of the same code. Stone et al. [57] implemented a linearly implicit fourth-order stiff Rosenbrock solver in the OpenCL for various platforms including CPUs, GPUs, and MICs. They found that SIMD vectorization improves integrator performance over an OpenMP baseline vectorized by simple compiler hints (i.e., `#pragmas`) by 2.5–2.8 × on the CPU and 4.7–4.9 × on the MIC, while the GPU performs only 1.4–1.6 × faster than the OpenMP baseline due to thread divergence [57].

### 3.1.2 Goals of this study

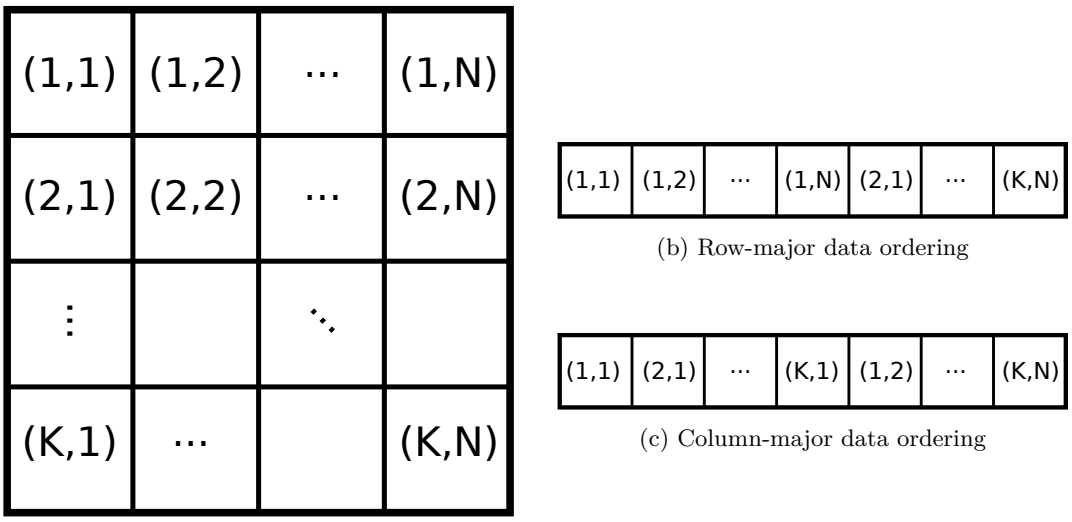
In this chapter we

- Derive and verify a new Jacobian formulation that greatly increases sparsity;
- Detail the implementation of cross-platform SIMD/SIMT vectorization for CPUs, GPUs, and other accelerators;
- Investigate the performance of SIMD/SIMT-vectorization for a wide range of chemical kinetic models, and compare with the previous version of our analytical chemical kinetic Jacobian code [44]; and finally
- Discuss future extensions to this work as well as several promising directions for SIMD/SIMT vectorization in reactive-flow simulations.

This work builds upon our previous analytical chemical kinetic Jacobian code, `pyJac` [44], using the new formulation, `pyJac v2`, to achieve these goals. To our knowledge is the first open-source, verified effort that vectorizes the evaluation of chemical-kinetic source terms and Jacobian matrices for any chemical model on a wide selection of platforms.

## 3.2 Methodology

### 3.2.1 Data ordering and vectorization patterns




(a) A simple 2-D data array with K rows and N columns.

(b) Row-major data ordering

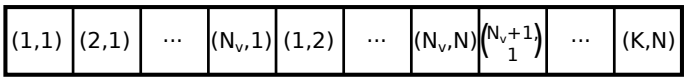
(c) Column-major data ordering

Figure 3.2: Simple data-layout patterns for 2-D arrays

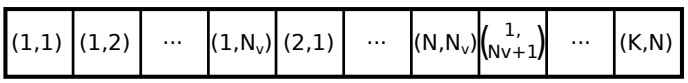
When storing arrays for a chemical kinetic model, the data-storage layout and vectorization patterns are critical to achieving high-performance code. Figure 3.2a depicts an example data array with $K$ rows and $N$ columns where index $(i, j)$ corresponds to the $i$th row and $j$th column. For example, the concentration of species $j$ for the $i$th thermochemical state would be stored in $[C]_{i,j}$ with $1 \leq i \leq N_{\text{state}}$ (the number of thermochemical states considered for evaluation) and $1 \leq j \leq N_{\text{sp}}$ (the number of species in the model). The stored concentrations would then have $K = N_{\text{state}}$ rows and $N = N_{\text{sp}}$ columns.

The "C" (C row-major) format stores the concentrations of all species for a single thermochemical condition $i$ sequentially in memory, i.e., with $[C]_{1,1}$ in index 1 (using one-based index notation), $[C]_{1,2}$ in index 2, and so on, as shown in Fig. 3.2b. Conversely, in the "F" (Fortran column-major) format the concentrations of a single species $j$ over all thermochemical states lie adjacent in memory, corresponding to storing $[C]_{1,1}$ in index 1, $[C]_{2,1}$ in index 2, and so on, as shown in Fig. 3.2c. This ordering strongly affects the performance of SIMD/SIMT-vectorized algorithms, as does the device (CPU, GPU, etc.) and vectorization pattern in question.

In a *shallow* SIMD/SIMT vectorization (also referred to as "per-thread" in previous works using GPUs [51]), each SIMD lane or SIMT thread in a compute unit evaluates the source terms or Jacobian for a different thermochemical state. If the data is stored in "F"-order, the SIMD lanes/SIMT thread accessing $[C]_{1,j} \dots [C]_{N_v,j}$ will load sequential locations in memory, where $[C]_{i,j}$ is the concentration of species $j$ for state $i$ and $N_v$ is the SIMD vector-width or the number of threads in a SIMT warp. The first $(j+1)$th species concentration, $[C]_{1,j+1}$, will be $N_{\text{state}}$ memory locations away; this increases the likelihood of cache misses on the CPU [120], but conversely well matches the pattern of coalesced memory access on the GPU [121].
In a *deep* SIMD/SIMT vectorization (also referred to as "per-block" in previous GPU works [51, 110]), a compute unit uses its SIMD lanes/SIMT threads cooperatively to evaluate the thermochemical source terms for a single thermochemical state; thus SIMD lanes loading $[C]_{1,j} \dots [C]_{1,j+N_v}$ will access sequential memory locations if the data is stored in "C"-order. Further, in "C" ordering any two species concentrations within the same thermochemical state lie at most $N_{\text{sp}}$ locations away, with $N_{\text{sp}} \ll N_{\text{state}}$ in most cases; this greatly improved data locality increases the chances of a cache hit on the CPU, but may lead to uncoalesced memory accesses on the GPU. Deep vectorization requires synchronization between SIMD lanes/SIMT threads via memory fences/barriers, a potentially expensive operation. In addition, deep vectorization may result in SIMD waste or SIMT thread divergence caused by different lanes/threads executing different instructions (e.g., resulting from different if/then branches). Shallow vectorization may also experience SIMD waste or SIMT thread divergence, e.g., in chemical kinetic integration due to varying internal solver time-step sizes [110]. However, in this work shallow vectorization is largely unaffected by this concern as the only major code paths that differ between vector lanes are high/low-temperature polynomial evaluations and differing pressures for P-Log reactions, which cause far fewer issues compared with differing internal ODE integration time-steps [110].




(a) Row-major, shallow-vectorized data ordering




(b) Column-major, deep-vectorized data ordering

Figure 3.3: Vectorized data-ordering patterns

Finally, Fig. 3.3 shows a vectorized data-ordering that improves the caching patterns of a shallow, "C"-ordered SIMD vectorization on the CPU (Fig. 3.3a) and a deep, "F"-ordered SIMT vectorization on the GPU (Fig. 3.3b). We accomplish this by splitting the slower-varying axis of

the data array—columns for "C"-ordering, and rows for "F"-ordering—into chunks of size $N_v$ (the SIMD vector width or SIMT warp size) and laying these data out contiguously in memory. For example, using the shallow-vectorized "C"-ordering pictured in Fig. 3.3a, the concentrations of species $j$ for states $i$ to $i + N_v$ ($[C]_{i,j}, \ldots, [C]_{i+N_v,j}$) lie contiguously in memory and are followed by the concentrations of species $j + 1$ for the same states ($[C]_{i,j+1}, \ldots, [C]_{i+N_v,j+1}$). This pattern ensures that any SIMD operation occurs on data contiguous in memory, which greatly improves caching and SIMD throughput; it is also similar to OpenCL's native vector data-types, e.g., `double8` treats eight contiguous double-precision floating-point numbers as a single vector datum. Conversely, the data-ordering in Fig. 3.3b enables coalesced memory accesses for "F"-ordered, deep SIMT vectorization on the GPU. We will discuss the effects of these various data-ordering and vectorization patterns on performance in Section 3.3.5.

### 3.2.2 Thermochemical source terms and Jacobian

This new version of `pyJac` is capable of evaluating the thermochemical source-terms for using the constant-pressure (CONP) or constant-volume (CONV) assumption*. In this section, we will outline a brief summary of the system evaluated by `pyJac`; Appendix C contains the complete—and lengthy—derivations.

The thermochemical state vector consists of the temperature, a non-constant thermodynamic state parameter (volume or pressure for CONP and CONV, respectively), and the number of moles of all species except the last species in the chemical model, typically taken to be the bath gas (e.g., $N_2$):

$$\Phi = \left\{T, V, n_1, n_2 \ldots n_{N_{\text{sp}}-1}\right\} \qquad \text{for CONP,} \tag{3.1a}$$

$$\Phi = \left\{T, P, n_1, n_2 \ldots n_{N_{\text{sp}}-1}\right\} \qquad \text{for CONV,} \tag{3.1b}$$

where $T$ is the temperature, $V$ and $P$ the volume and pressure respectively, and $n_j$ the number of moles of the $j$th species in the model (containing $N_{\text{sp}}$ total species).

This state vector—inspired by Schwer et al. [101]—has a number of beneficial features. First, the state vector results in highly sparse chemical kinetic Jacobians, as will be detailed in Section 3.3.4. Second, this formulation explicitly conserves mass, because the number of moles and rate of change of the final species are calculated from the ideal gas law and conservation of mass, respectively; see Appendix C for the full details of the governing equations. The system is not over-constrained [116] and does not require use of a more-complicated differential algebraic

*Note: in this context, the "constant-pressure" and "constant-volume" assumptions refer to evaluation within a reaction sub-step in the operator splitting scheme, rather than a general constant-pressure or constant-volume reactive-flow simulation.

equation solver (as compared to an ODE integrator) for integration. Finally, the chemical kinetic Jacobian for this formulation changes relatively little between the CONP and CONV forms, making maintaining the codebase much simpler. Although most current combustion codes do not use species moles as a state variable, conversion to/from the more-common mass/mole fractions and moles is straightforward, and the choice of variables no longer matters once inside the integration of an chemical kinetic initial-value problem (IVP).

The evolution of the thermochemical state vector is described by a set of chemical kinetic ordinary differential equations:

$$f = \frac{\mathrm{d}\Phi}{\mathrm{d}t} = \left\{ \frac{\mathrm{d}T}{\mathrm{d}t}, \frac{\mathrm{d}V}{\mathrm{d}t}, \frac{\mathrm{d}n_1}{\mathrm{d}t}, \frac{\mathrm{d}n_2}{\mathrm{d}t} \ldots \frac{\mathrm{d}n_{N_{\mathrm{sp}}-1}}{\mathrm{d}t} \right\} \qquad \text{for CONP,} \tag{3.2a}$$

$$f = \frac{\mathrm{d}\Phi}{\mathrm{d}t} = \left\{ \frac{\mathrm{d}T}{\mathrm{d}t}, \frac{\mathrm{d}P}{\mathrm{d}t}, \frac{\mathrm{d}n_1}{\mathrm{d}t}, \frac{\mathrm{d}n_2}{\mathrm{d}t} \ldots \frac{\mathrm{d}n_{N_{\mathrm{sp}}-1}}{\mathrm{d}t} \right\} \qquad \text{for CONV.} \tag{3.2b}$$

For both CONP and CONV, the molar source terms are [122]:

$$\frac{\mathrm{d}n_k}{\mathrm{d}t} = V\dot{\omega}_k \qquad k = 1, \ldots, N_{\mathrm{sp}} - 1, \tag{3.3}$$

where $\dot{\omega}_k$ is the $kth$ species' overall molar production rate:

$$\dot{\omega}_k = \sum_{i=1}^{N_{\mathrm{reac}}} \nu_{k,i} R_i c_i \ , \tag{3.4}$$

$\nu_{k,i}$ is the net stoichiometric coefficient of species $k$ in reaction $i$, $N_{reac}$ is the total number of reactions, $R_i$ is the net rate of progress of reaction $i$, and $c_i$ is the pressure-dependent modification term, i.e., for third-body or falloff/chemically-activated reactions. `pyJac` is capable of evaluating all modern reaction types, e.g., P-Log and Chebyshev reactions.

The temperature source-term [122] is:

$$\frac{\mathrm{d}T}{\mathrm{d}t} = -\frac{\sum_{k=1}^{N_{\mathrm{sp}}} H_k \dot{\omega}_k}{\sum_{k=1}^{N_{\mathrm{sp}}} [C]_k C_{p,k}} \qquad \text{for CONP,} \tag{3.5a}$$

$$\frac{\mathrm{d}T}{\mathrm{d}t} = -\frac{\sum_{k=1}^{N_{\mathrm{sp}}} U_k \dot{\omega}_k}{\sum_{k=1}^{N_{\mathrm{sp}}} [C]_k C_{v,k}} \qquad \text{for CONV,} \tag{3.5b}$$

where $H_k$, $U_k$, $C_{p,k}$, and $C_{v,k}$ are the enthalpy, internal energy, constant-pressure specific heat, and constant-volume specific heat of species $k$ in molar units, respectively, while $[C]_k$ is the concentration, given by

$$[C]_k = \frac{n_k}{V} \ . \tag{3.6}$$

By differentiating the ideal gas law, given by

$$PV = n\mathcal{R}T \tag{3.7}$$

where $\mathcal{R}$ is the ideal-gas constant in molar units, we find the volume and pressure source terms (where $W_k$ and $W_{N_{\text{sp}}}$ are the molecular weights of species $k$ and $N_{\text{sp}}$, respectively):

$$\frac{\mathrm{d}V}{\mathrm{d}t} = V\left(\frac{T\mathcal{R}}{P}\sum_{k=1}^{N_{\text{sp}}-1}\left(1-\frac{W_k}{W_{N_{\text{sp}}}}\right)\dot{\omega}_k + \frac{1}{T}\frac{\mathrm{d}T}{\mathrm{d}t}\right) \qquad \text{for CONP,} \tag{3.8a}$$

$$\frac{\mathrm{d}P}{\mathrm{d}t} = T\mathcal{R}\sum_{k=1}^{N_{\text{sp}}-1}\left(1-\frac{W_k}{W_{N_{\text{sp}}}}\right)\dot{\omega}_k + \frac{P}{T}\frac{\mathrm{d}T}{\mathrm{d}t} \qquad \text{for CONV.} \tag{3.8b}$$

`pyJac` arranges the computed Jacobian entries such that entry $(i,\, j)$ corresponds to the partial derivative of the $i$th source-term in Eq. (3.2) by the $j$th state variable in Eq. (3.1):

$$\mathcal{J}_{i,j} = \frac{\partial f_i}{\partial \Phi_j}, \qquad i, j = 1 \ldots N_{\text{sp}} + 1\ . \tag{3.9}$$

Appendix C for this article contains the complete derivation of the Jacobian used by `pyJac`, for interested readers.

### 3.2.3 Code generation and testing infrastructure

The new version of `pyJac` uses the Python package `loo.py` [123] for code generation, which translates pseudo-code and data to OpenCL/C code. As the name implies, `loo.py` generates code using `for` loops; this differs from the previous version of `pyJac` [124] that generates static code—i.e., fully unrolled loops, with thermodynamic/reaction parameters written directly in code rather than stored in arrays. In our previous work [44], this static code generation caused some issues with large file sizes, long compilation times, and even occasionally broke the `gcc` and `nvcc` compilers (the latter issue necessitated splitting the Jacobian/source-term evaluations into separate files). We will discuss the implications of this change in Section 3.3.5.2, where the performance of the new version of `pyJac` will be compared with the previous version.
In addition, `loo.py` allows the user to more easily make changes to the structure of the generated program, e.g., the data ordering, vectorization, and threading patterns, as well as switch the target language for code generation (and more simply extend to additional languages, e.g., CUDA). Further, `loo.py` can execute developed subroutines from Python (natively for C code, or via `PyOpenCL` [125] for OpenCL), enabling unit testing/verification for each component of the Jacobian or source terms; the unit testing suite also helps ensure that bugs are not present in less commonly used code-paths, and are not introduced by future code changes. The source terms

and sub-components thereof (e.g., rates of progress, pressure-modification terms) are directly compared with Cantera [126], while the automatic differentiation code Adept [127, 128] provides reference values for Jacobian sub-components. We use the Portable OpenCL (POCL) implementation [129] and OpenMP [130] to perform OpenCL and C unit testing, respectively, on the continuous-integration framework Travis CI [131]. We will discuss verification of the complete (as opposed to the sub-component testing discussed here) generated source-terms and Jacobian codes in detail in Section 3.3.2.

## 3.3 Results and discussion

### 3.3.1 Testing platforms

| CPU Model | Xeon X5650 | E5-2690 V3 |
|---|---|---|
| Instruction Set | SSE4.2 | AVX2 |
| Vector Width | two doubles | four doubles |
| Cores | $2 \times 6$ | $2 \times 12$ |
| Identifier | `sse4.2` | `avx2` |
| OpenCL Version | 1.2 | 1.2 |

Table 3.1: The Intel CPU configurations used in this study. The vector widths are reported in (ideal) number of double operations per SIMD instruction, as this will be used in measuring SIMD efficiency; for reference, the vector widths of the `sse4.2` and `avx2` machines are 128 bit and 256 bit, respectively. The identifier field will be used as a shorthand descriptor in the performance plots to quickly identify the CPU type.

We ran the performance and verification studies for this work on a variety of CPU and GPU platforms. Table 3.1 shows the number of cores, vector instruction set, and model of the CPUs used in this work; each CPU had both `v16.1.1` of the Intel OpenCL runtime [132] and `v1.0` of the POCL [129] runtime installed, both enabling OpenCL `v1.2` execution. Additionally, `v5.0` of the LLVM/`clang` [133] compiler chain was installed on all machines to enable use of POCL. Table 3.2 lists the model, number of CUDA cores, and Nvidia driver of each GPU we used. Nvidia's OpenCL runtime is bundled with the Nvidia driver [121], hence the driver version is used to specify the OpenCL runtime version.

| Nvidia Model | Tesla C2075 | Tesla K40m |
|---|---|---|
| Driver Version | 384.81 | 387.26 |
| CUDA Cores | 448 | 2880 |
| Identifier | `C2075` | `K40m` |
| OpenCL Version | 1.1 | 1.2 |
| Memory* | 6 GB | 12 GB |

Table 3.2: The Nvidia GPU configurations used in this study. Nvidia's OpenCL runtime is provided with the graphics driver, rather than any specific version of CUDA. The identifier field will be used as a shorthand descriptor during analysis of the performance results.

Table 3.3 lists the platforms and vectorization/execution patterns that they are capable of running. The Intel and Nvidia OpenCL runtimes lack implementations of atomic operations on double-precision variables; `pyJac` currently needs these to run deep-vectorized code. On the other hand, POCL is an open-source OpenCL runtime that works on all CPU types tested here, and does implement these atomic operations. However, POCL's implicit vectorization module—which uses the LLVM compiler [133] to translate OpenCL code to vectorized machine code—typically fails to achieve much, if any, speedup. Thus POCL is useful for verification but not necessarily for performance studies; it is noted that while POCL is currently used by `pyJac` for unit-testing purposes, it is not required to use `pyJac`. We will expand upon this discussion in Section 3.5 to highlight future directions.

| Platform | Parallel | Shallow Vectorization | Deep Vectorization |
|---|---|---|---|
| OpenMP | ✓ | – | – |
| POCL OpenCL | ✓ | ✓ | ✓ |
| Intel OpenCL | ✓ | ✓ | – |
| Nvidia OpenCL | – | ✓ | – |

Table 3.3: The platforms used in this study and the execution /vectorization patterns that they are capable of running.

Finally, Table 3.4 displays the chemical kinetic models used in this work, as well as number of partially stirred reaction conditions (PaSR) used in the condition database for each. Our previous works describe the creation of the PaSR databases in detail works [44, 110].

| Model | Number of Conditions | Reference |
|---|---|---|
| $H_2$/CO | 900,900 | [88] |
| GRI-Mech 3.0 | 450,900 | [89] |
| USC-Mech II | 91,800 | [135] |
| $iC_5H_{11}OH$ | 450,900 | [136] |

Table 3.4: The chemical kinetic models used in this study and number of conditions in the partially stirred reactor database for each.

### 3.3.2 Source-term verification

We verified the reaction rates of progress (ROP), species production rates, and temperature rates in this study by comparing with values calculated using Cantera [126]. However, special care must be taken due to floating-point arithmetic issues.

For a direct comparison, a relative error norm of a quantity $X_{ij}$ over all states $j$ and reactions (or

*A driver implementation issue limited total memory to 4 GB and 10 GB on the `C2075` and `K40m` GPUs, respectively [134].

species) $i$ was computed using the $L^\infty$ norm:

$$E_X = \left\| \frac{|X_{ij,\mathrm{CT}} - X_{ij}|}{10^{-10} + 10^{-6} \times |X_{ij,\mathrm{CT}}|} \right\|_\infty , \tag{3.10}$$

where the CT subscript indicates values from Cantera [126].

However, computing the net ROP of reaction $i$ for state $j$ from the forward and reverse ROP, i.e., $R_{ij} = R'_{ij} - R''_{ij}$, can easily lose accuracy as the net ROP may be many orders of magnitude smaller than the forward and/or reverse rates—particularly near chemical equilibrium. To quantify this phenomenon, we first define the error in forward ROP as

$$\varepsilon'_{ij} = \left| R'_{ij} - R'_{ij,\mathrm{CT}} \right| , \tag{3.11}$$

while the error in reverse ROP, $\varepsilon''_{ij}$, can be defined analogously. Finally, for the reaction $i^*$ and the state $j^*$ that result in the largest error in net ROP, i.e., $E_R$, an estimate of the error attributable to floating-point error accumulation from the forward and reverse ROPs can be obtained using

$$E_\varepsilon = \frac{\max(\varepsilon'_{i^*j^*}, \varepsilon''_{i^*j^*})}{10^{-10} + 10^{-6} \times |R_{i^*j^*,\mathrm{CT}}|} . \tag{3.12}$$

This estimate allows for directly comparing the error in forward or reverse ROPs with the value of the net ROP itself; the error in net ROP will be large if these are similar in magnitude.

| Model | $H_2$/CO | GRI-Mech. 3.0 | USC-Mech II | $iC_5H_{11}OH$ |
|---|---|---|---|---|
| $E_{R'}$ | $1.56 \times 10^{-8}$ | $2.95 \times 10^{-8}$ | $9.42 \times 10^{-8}$ | $4.86 \times 10^{-4}$ |
| $E_{R''}$ | $6.92 \times 10^{-8}$ | $6.53 \times 10^{-8}$ | $1.20 \times 10^{-7}$ | $5.07 \times 10^{-4}$ |
| $E_R$ | $1.49 \times 10^{1}$ | $1.11 \times 10^{0}$ | $2.80 \times 10^{0}$ | $4.82 \times 10^{-1}$ |
| $E_\varepsilon$ | $1.48 \times 10^{1}$ | $1.13 \times 10^{0}$ | $2.93 \times 10^{0}$ | $5.03 \times 10^{-1}$ |
| $E_{\frac{\mathrm{d}n}{\mathrm{d}t}}$ | $2.53 \times 10^{1}$ | $2.60 \times 10^{0}$ | $7.62 \times 10^{0}$ | $1.58 \times 10^{1}$ |
| $E_{\frac{\mathrm{d}T}{\mathrm{d}t}}$ | $3.94 \times 10^{5}$ | $3.35 \times 10^{8}$ | $3.95 \times 10^{6}$ | $7.11 \times 10^{7}$ |
| $E_{\frac{\mathrm{d}S}{\mathrm{d}t}}$ | $3.52 \times 10^{12}$ | $3.46 \times 10^{12}$ | $3.44 \times 10^{12}$ | $3.38 \times 10^{12}$ |

Table 3.5: Summary of errors in rates of progress, species, temperature, and thermodynamic state-parameter rate compared with Cantera. Error statistics are based on the infinity-norm of the relative error detailed in Eq. (3.10) for each quantity. The "S" in $E_{\frac{\mathrm{d}S}{\mathrm{d}t}}$ refers to the thermodynamic state parameter, either $V$ or $P$ for CONP and CONV, respectively.

Table 3.5 compares `pyJac` v2's source-term evaluations with Cantera's [126] using the data set of PaSR conditions (Table 3.4). The forward and reverse ROPs agree closely for all models, though the error norm is ~3–4 orders of magnitude larger for the isopentanol model. This discrepancy results from differences in evaluation of P-Log reactions between `pyJac` and Cantera: `pyJac` computes the logarithm of the upper and lower reaction Arrhenius rates analytically (see Appendix D) while Cantera evaluates this term numerically. If we neglect the errors from P-Log reactions in Eq. (3.10), the errors for the forward and reverse ROPs fall to $5.44 \times 10^{-8}$ and

$1.59 \times 10^{-7}$, respectively. This discrepancy does not imply any actual error in either `pyJac` or Cantera—in fact, the error still lies well within the proscribed tolerances in Eq. (3.10)—but merely emphasizes how even small code changes can affect the accumulation of floating-point errors.

The error in the net ROP further underscores this point: it is ~3–9 orders of magnitude (or 7–9 orders of magnitude when including P-Log reaction contributions) larger than the error in forward or reverse ROP. Table 3.5 shows that the magnitudes of $E_\varepsilon$ and $E_R$ agree in all cases, indicating that the accumulation of floating-point error from the forward and reverse ROPs causes this large increase in error as previously discussed. The magnitudes of the errors in molar species production rate and net ROP agree, but thermodynamic properties amplify the error in net species production rates and lead to high discrepancies in temperature and state-parameter rates. Again, these discrepancies in net ROP will not necessarily cause errors when integrating the chemical kinetics—either in `pyJac` or Cantera—as this loss of accuracy only occurs when the forward and reverse ROPs are nearly equal (i.e., near equilibrium).

### 3.3.3 Jacobian verification

As in our previous work [44], we determined Jacobian matrix correctness by comparing with that obtained by automatic differentiation of the `pyJac`-generated source term, using the `Adept` software library [127, 128]. We previously explained this choice fully [44], but broadly speaking automatic differentiation provides relatively fast, highly accurate Jacobian matrix evaluation with minimal additional programming effort. (In contrast, it is challenging to obtain robust, accurate Jacobians using finite differences.) The discrepancy between the analytical and automatic-differentiation Jacobians for thermochemical state $k$, denoted by $\mathcal{J}_k$ and $\hat{\mathcal{J}}_k$ respectively, is determined by the relative error Frobenius norm over all Jacobian indices $i, j$:

$$E_{\mathrm{rel},k} = \left\| \frac{\hat{\mathcal{J}}_{ij,k} - \mathcal{J}_{ij,k}}{\hat{\mathcal{J}}_{ij,k}} \right\|_F . \tag{3.13}$$

To avoid large relative discrepancies in small nonzero Jacobian elements due to accumulation of floating-point error, the Frobenius norm of the automatically differentiated Jacobian is calculated over all thermochemical states $k$:

$$\mathcal{T} = \left\| \hat{\mathcal{J}} \right\|_F . \tag{3.14}$$

The error statistics reported in this section are then based only on matrix elements where $\mathcal{J}_{ijk} \geq \frac{\mathcal{T}}{\mathcal{C}}$, where $\mathcal{C}$ is a tunable threshold parameter; this filtered form of Eq. (3.13) is denoted $E_{\mathcal{C},k}$. Finally, the Frobenius norm is calculated over all the states $k$ in the PaSR thermochemical

condition database:

$$E_{\mathcal{C}} = \|E_{\mathcal{C},k}\|_F \ . \tag{3.15}$$

This error norm is quite different from the relative error Frobenius norm suggested by Anderson et al. [65] for quantifying the error of matrices in LAPACK, e.g., over Jacobian indices $i, j$:

$$E_{\mathcal{L},k} = \frac{\left\|\hat{\mathcal{J}}_{ijk} - \mathcal{J}_{ijk}\right\|_F}{\left\|\hat{\mathcal{J}}_{ijk}\right\|_F}$$

and states $k$:

$$E_{\mathcal{L}} = \|E_{\mathcal{L},k}\|_F \ . \tag{3.16}$$

In our experience, the accuracy of larger elements in a Jacobian often dominates the LAPACK error norm, while the filtered error norm can identify errors in both large and small Jacobian entries. Further, with the tunable threshold parameter $\mathcal{C}$, we can assess the error of different ranges of element sizes and isolate the effects of floating-point error. For reference, both our error norm and the LAPACK error norm will be reported.

| Model | $E_{\mathcal{L}}$ | $\bar{\mathcal{T}}$ | $E_{\mathcal{C}=10^{20}}$ | $E_{\mathcal{C}=10^{15}}$ |
|---|---|---|---|---|
| $H_2$/CO | $1.862 \times 10^{-14}$ | $6.431 \times 10^{18}$ | $1.741 \times 10^{0}$ | $4.508 \times 10^{-5}$ |
| GRI-Mech 3.0 | $1.567 \times 10^{-14}$ | $7.783 \times 10^{19}$ | $3.842 \times 10^{-7}$ | $3.687 \times 10^{-7}$ |
| USC-Mech II | $1.137 \times 10^{-14}$ | $2.830 \times 10^{21}$ | $1.199 \times 10^{-2}$ | $1.983 \times 10^{-7}$ |
| $iC_5H_{11}OH$ | $1.227 \times 10^{-10}$ | $2.733 \times 10^{26}$ | $1.363 \times 10^{-3}$ | $2.764 \times 10^{-5}$ |

Table 3.6: Summary of Jacobian matrix verification results. The reported error statistics are the maximum filtered relative error $E_{\mathcal{C}}$ and LAPACK error $E_{\mathcal{L}}$ over all test platforms, vectorization patterns (Table 3.3), CONP/CONV, and sparse/dense Jacobians. The Frobenius norm described in Eq. (3.14) varies slightly between the CONP and CONV cases; the reported $\bar{\mathcal{T}}$ is the average of the two, with the appropriate value used during calculations of the error statistics.

Table 3.6 reports the maximum $E_{\mathcal{C}}$ and $E_{\mathcal{L}}$ values over all test platforms and vectorization patterns (see Table 3.3), sparse and dense (see Section 3.3.4), as well as CONP and CONV formulations. The most stringent filtered error norm ($\mathcal{C} = 10^{20}$) ranges from $10^{-7}$–$10^{0}$; the largest error is for the $H_2$/CO model. For this model, $\mathcal{T}$ is smaller than the tolerance of $10^{20}$, and hence the error norm considers Jacobian entries smaller than $\mathcal{O}(1)$. GRI-Mech 3.0 has a $\mathcal{T}$ roughly an order of magnitude larger and so the stringent error norm is significantly smaller: $\mathcal{O}(10^{-7})$. Given the intricacy of floating-point error evaluation, the use of different languages and OpenCL platforms (the effect of these differences will be explored in Appendix D), and the general complexity of `pyJac` it would be exceedingly difficult to pinpoint an exact cause for this phenomenon, as was done in Section 3.3.2. To ensure no bugs or errors present in the Jacobians generated by `pyJac`, relaxed filtered error norms ($\mathcal{C} = 10^{15}$) are also presented for each model

in Table 3.6. This relaxed norm is smaller by 2–5 orders of magnitude for all models—except GRI-Mech 3.0, where the stringent case already has small error as previously discussed—which indicates that accuracy is higher when the smaller Jacobian entries are excluded. This result suggests that floating-point error accumulation controls the stringent filtered error norm.
The relative LAPACK error norm—ranging from ~$10^{-10}$–$10^{-14}$—re-enforces this finding, as it indicates roughly 10–14 digits of accuracy [65]. The $iC_5H_{11}OH$ model has the largest LAPACK error norm, likely due to the presence of P-Log/Chebyshev reactions and the resulting complicated derivatives with many logarithms, exponetiations, and summations. Further, the LAPACK error norm does not correlate well with the stringent filtered error norm, e.g., USC-Mech II has the smallest LAPACK error norm ($1.137 \times 10^{-14}$) but the second-largest stringent filtered error norm ($1.119 \times 10^{-2}$). Conversely, the model with the largest LAPACK error norm, $iC_5H_{11}OH$ has the second smallest stringent filtered error norm: $E_{\mathcal{L}} = 1.227 \times 10^{-10}$ and $E_{\mathcal{C}=10^{20}} = 1.363 \times 10^{-3}$, again suggesting that floating-point error accumulation influences the stringent error norm. These findings, along with the individual unit-testing of all chemical source-terms and Jacobian sub-components described in Section 3.2.3, gives high confidence in the correctness of `pyJac` v2.

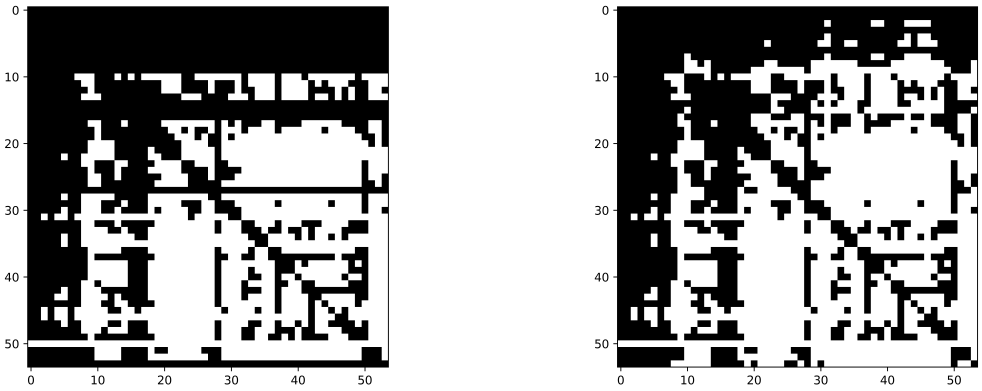
### 3.3.4 Sparsity patterns

In general, the Jacobian matrices generated by `pyJac` are largely sparse with non-zero entries corresponding to species that participate in the same reaction or non-default efficiency third-body species in a reaction, with dense rows/columns corresponding to temperature and the thermodynamic state parameter. However, the explicit-mass conservation formulation of `pyJac` can introduce additional non-zero entries in two ways. First, if the last species in the model (i.e., the bath gas) participates directly in any reaction, the derivative of its forward or reverse rate of progress is non-zero with respect to all other species in the model, regardless of whether the other species participate in that reaction or not. Similarly, if the last species has a third-body efficiency not equal to the default (one), this will again create nonzero derivatives for the pressure-modification term with respect to all other species (see Appendix C). Either case will result in a fully dense Jacobian row for all species with a non-zero net stoichiometric coefficient in such a reaction.
However, `pyJac` v2 allows the user to ignore these derivatives (via a command-line switch) and avoid the adverse effects on Jacobian sparsity.* The rationale behind this choice is that many common implicit integration techniques (e.g., CVODE [137]) used to solve chemical-kinetic initial-value problems are formulated around the assumption that the supplied Jacobian is

*Alternatively, one may choose the last species as one that does not participate in any reactions.

approximate; this allows the Jacobian and its LU factorization to be reused for multiple internal integration time steps, accelerating the solution process. Such solvers do not need the exact form of the Jacobian and thus the so-called "approximate" form is preferable. Though this might be used as a crude form of preconditioning for such solvers, the primary purpose is merely to increase Jacobian sparsity; McNenly et al. [70] more thoroughly investigated preconditioners. Hence, in this section we will detail the sparsity of both forms of the Jacobian for the chemical models tested.

Figure 3.4 graphically represents the Jacobian sparsity of GRI-Mech 3.0. In particular we note that Fig. 3.4b has several rows that are no longer fully dense, as result of its approximate form; these rows correspond to species directly interacting with $N_2$, largely in GRI-Mech 3.0's nitrogen chemistry reactions. Table 3.7 shows the density of the exact and approximate Jacobians for all chemical kinetic models tested in this work. The smallest model, $H_2$/CO, is very dense with 71.4 % of the exact Jacobian entries non-zero; this drops to 56.7 % for GRI-Mech 3.0, continues to decrease to 28.2 % for USC-Mech II, and is just 11.5 % for the isopentanol model. The approximate Jacobian assumption drops the density of Jacobian by ~3–7 % for all models.




(a) The "exact" Jacobian.

(b) The "approximate" Jacobian.

Figure 3.4: A graphical representation of the sparsity pattern of the chemical kinetic Jacobian generated by `pyJac` for GRI-Mech 3.0. Black squares indicate a non-zero Jacobian entry, while white square correspond to an empty entry. The numbers indicate the index of the entry in the state vector.

Currently, `pyJac` can use two common sparse-matrix storage formats [138]: compressed row storage (CRS) and compressed column storage (CCS), used for "C" and "F"-ordered data respectively. For brevity we will outline only the CRS format but the CCS format is similar [138]. An $N \times N$ CRS matrix is stored using three vectors: a value vector of length $N_{NZ}$ (the number of non-zero elements in the Jacobian) that stores the elements of the Jacobian, a row pointer vector of length $N$ that stores the locations in the value vector that begin a row, and a column index vector length $N_{NZ}$ that stores the column indices of the elements in the value vector. The Jacobian access pattern used by `pyJac` is fairly irregular; for simplicity we will only discuss

| Model | Exact Jacobian Density | Approximate Jacobian Density |
|---|---|---|
| $H_2/CO$ | 71.4 % | 68.4 % |
| GRI-Mech 3.0 | 56.7 % | 49.8 % |
| USC-Mech II | 28.2 % | 26.4 % |
| $iC_5H_{11}OH$ | 11.5 % | 7.98 % |

Table 3.7: The density of the exact and approximate Jacobians generated by `pyJac` for the various models studied.

looping-structure of species derivatives calculations since these form the bulk of the computation and have the most challenging Jacobian access patterns. In general, an outer loop iterates over all reactions of a certain type (e.g., falloff reactions) and calculates the relevant Jacobian subproducts—independent of any particular species—for the reaction (e.g., the derivative of the falloff pressure modification term). Two inner loops then iterate over the species in a reaction, updating the Jacobian entries for these species as appropriate. This pattern leads to fairly easily vectorizable code and efficient Jacobian evaluation, since the bulk of the computation depends only on the reaction in question, as discussed in our previous work [44]. Generally, this means that a lookup operation is required to find the sparse Jacobian index for any pair of state variables; in some cases this can be avoided, e.g., the rows corresponding to derivatives of the temperature and thermodynamic state parameter source-terms are fully dense in `pyJac`, and hence no lookup is necessary. This lookup operation is currently implemented as a simple "for" loop, e.g., for a sparse lookup of a pair of indices $(i, j)$ in a CRS matrix, the lookup function searches the column index vector between the values $\texttt{row\_pointer}[i], \ldots, \texttt{row\_pointer}[i+1]$ for $j$, and returns the offset from $\texttt{row\_pointer}[i]$ (or $-1$ if not found). As will be explored in Section 3.3.5.2, this slows down sparse Jacobian evaluation, and might be improved by a static mapping of the full Jacobian indices to the sparse index (or some null value if the entry is empty). However, this would require increased constant-data usage, a limitation for OpenCL. Additionally, this might be an excellent usage of OpenCL's Image memory type (similar to texture memory in CUDA terminology). Both of these sparse indexing techniques merit future investigation.

### 3.3.5 Performance

The performance studies in this work were run on the platforms listed in Tables 3.1 and 3.2. Run times in each case were averaged over ten runs, each using the same set of PaSR conditions used in verification. The OpenMP Jacobian/source-term kernels, as well as the OpenMP/OpenCL wrapping code (responsible for initializing/transferring memory, reading input, etc.) was compiled with `gcc v5.4.0` on the `avx2/K40m` platforms and `gcc v4.8.5` on the `sse4.2/C2075` machines. The optimization level "`-O3 -mtune=native`" was used and no "fast math" OpenCL

optimizations were enabled. Additionally, the exact form of the Jacobian (as opposed the "approximate" form discussed in Section 3.3.4) was used in all cases. Finally, unless stated otherwise: the performance results used a single CPU core, the CONP assumption, a vector width of 8/128, and "C"/"F"-ordered data for the CPU/GPU cases, respectively; the run times reported are for the number of conditions specified in Table 3.6 and include data-transfer overhead to/from internal buffers used in `pyJac`. The effects of choice of vector width, data ordering, and differences between CONP and CONV evaluations on the CPU/GPU will be explored in Section 3.3.5.1 while parallel scaling for multiple CPU cores will be examined in Sections 3.3.5.1 and 3.3.5.2.

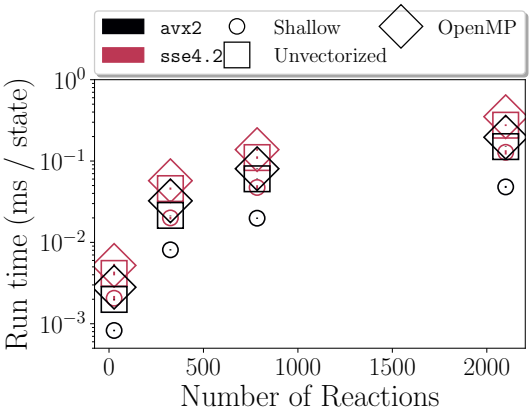
#### 3.3.5.1 Source-term evaluation

Figure 3.5 explores the performance of the source-term evaluations generated by `pyJac` on the CPU test platforms listed in Table 3.1. Source-term evaluations—critical in their own right for direct numerical simulations of reactive-flows [47], among other applications—also provide a convenient platform to detail the effects of various code configuration options before investigating the more involved Jacobian evaluation performance.

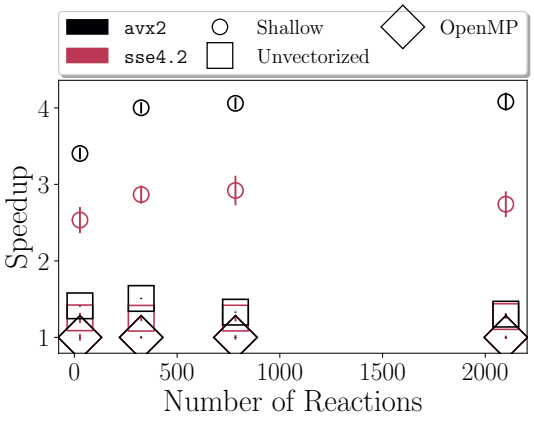
Figure 3.5a shows the mean run time per initial condition for both the `avx2` and `sse4.2` CPUs, using Intel OpenCL and OpenMP. This normalization of the run time by the number of initial conditions tested is chosen to account for the varying numbers of conditions in the PaSR databases for each model (Table 3.4). For both CPUs, the OpenMP implementation is the slowest for all models; interestingly, the unvectorized (i.e., strictly parallel) Intel OpenCL code is slightly faster than OpenMP in all cases. As expected, the `avx2` machine is faster than the `sse4.2` CPU for the strictly parallel cases, performing 1.82–2.13 × and 1.72–1.85 × faster for the unvectorized OpenCL case and OpenMP, respectively. Additionally, the shallow-vectorized OpenCL code performs significantly faster than either the OpenMP or unvectorized OpenCL codes on both processors.

Figure 3.5b details the extent of this speedup; the speedup displayed is calculated per-machine, e.g., the `avx2` shallow-vectorized code speedup is relative to OpenMP on the same CPU. On both machines, the unvectorized OpenCL code is faster than the baseline parallel OpenMP code, by 1.30–1.51 × on the `avx2` CPU and 1.25–1.27 × on the `sse4.2` machine. Additionally, the shallow-vectorized OpenCL code is 2.53–2.92 × and 3.40–4.08 × faster than the OpenMP code for the `sse4.2` and `avx2` machines, respectively.

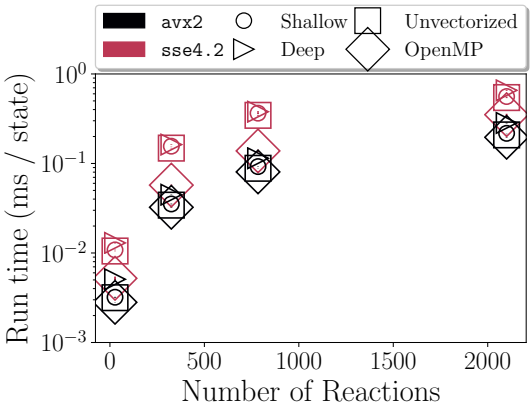
In contrast, Fig. 3.5c shows the mean run time per condition of deep, shallow, and unvectorized OpenCL codes using the POCL runtime, as compared with OpenMP parallelization. No speedup is achieved for either vectorization type on either CPU—indeed, the OpenMP case is faster on




(a) The mean run time per condition for each chemical model using the Intel OpenCL and OpenMP on both CPUs studied.




(b) The speedup achieved over the baseline OpenMP parallelization by both the unvectorized and shallow-vectorized Intel OpenCL codes; the speedup is presented on per-machine basis.




(c) The mean run time per condition of the Portable OpenCL runtime compared to OpenMP parallelization.

Figure 3.5: Mean run times per condition and speedups achieved by the various CPU OpenCL runtimes compared to OpenMP parallelization for each chemical model studied. The names in the legends correspond to the identifiers listed in Table 3.1.

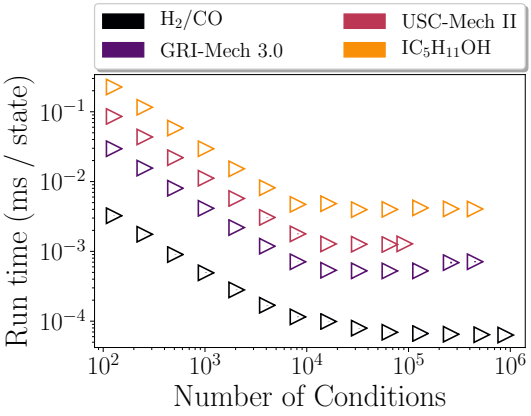
both CPUs, though we stress that Intel OpenCL runtime achieves vectorization using the same code and hardware. The reasons for this lack of vectorization with POCL are quite technical; however, personal communication with the developers of POCL revealed that to achieve vectorization, changes are required to both the way POCL prepares LLVM intermediate representation code, as well as improvements to LLVM's loop-vectorizer itself [139].* As will be discussed in Section 3.4, we hope that using explicit vector types (to lessen demands on the LLVM-vectorization module) in combination with some of these changes might solve this issue, but for the moment POCL is still quite useful as a verification tool.

Figure 3.6 shows the performance of evaluating source terms on the GPUs listed in Table 3.2. Figure 3.6a investigates how the number of initial conditions evaluated affects the mean run time per condition on the `K40m` GPU; the run time decreases until around ~$10^4$ conditions for all chemical models, at which point the GPU becomes saturated and performance levels off. The performance plateaus slightly later for the $H_2$/CO model compared with the others. Figure 3.6b shows the speedup in source-term evaluation that the `K40m` GPU achieves over the `C2075` GPU for the maximum number of conditions in Fig. 3.6a with two vector widths (i.e., GPU block size), 64 and 128. The best `K40m` case with a vector width of 128 is 1.40–1.88 × faster than the slowest case (`C2075` with a vector-width of 64) depending on the chemical model in question. Figure 3.6b also shows that varying the vector-width minimally affects performance for most of the `K40m` and `C2075` cases; the GRI-Mech 3.0 and USC-Mech II models show the largest improvements with a vector width of 128: ~10–18 % for both GPUs. This is likely caused by higher occupancy on the GPUs [140] i.e., the percent of the CUDA cores that are utilized during computation. Occupancy on the GPU is typically less than 100 % due to hardware constraints, e.g., limited shared memory size or registers per warp [140], however, it is unclear exactly how Nvidia's OpenCL runtime balances the registers/warps per streaming-multiprocessor which affects occupancy in this case.

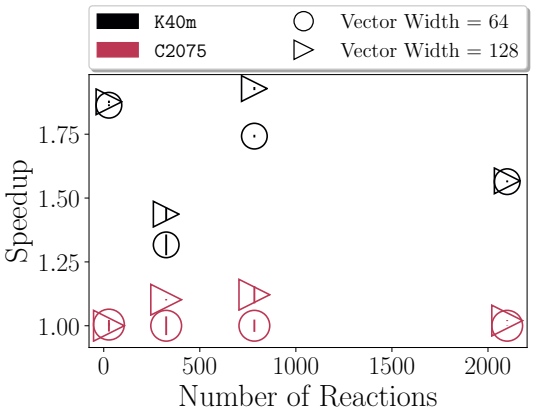
Figure 3.7 shows how changing data-ordering patterns, the CONP or CONV formulation, and the CPU vector width affect the performance of source-term evaluation in `pyJac`. Per Fig. 3.7a, we see that the choice of CONP or CONV formulation has little to no effect on run time for OpenMP as well as the shallow-vectorized/unvectorized Intel OpenCL codes on the `avx2` machine. Generally speaking, the difference between the CONP and CONV formulations only marginally affects performance regardless of CPU/GPU choice.

In contrast, Fig. 3.7b shows significant speedups of "C"-ordered data over "F"-ordered data on the `avx2` machine; the speedup presented is calculated per language, e.g., the 1.35–2.09 × speedup of the "C"-ordered OpenMP implementation is relative to the "F"-ordered OpenMP baseline.

*Specifically, improvements such as ensuring LLVM recognizes uniform vectorization loop bounds (even if said bounds *are* uniform in practice), proving vector instructions' ability to handle all edge cases identically to the corresponding scalar instruction, handing of branches and conditionals (potentially within POCL instructions themselves), and handling of memory access/vector-element extraction patterns.

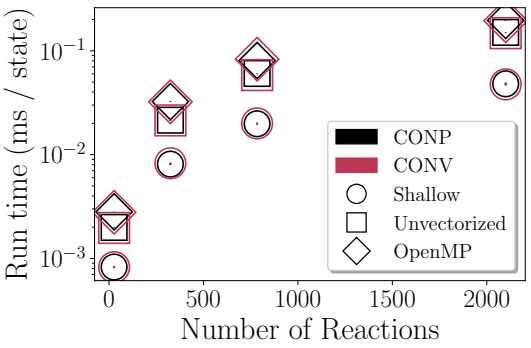



(a) The mean run time per condition for each chemical model on the `K40m` GPU as a function of the number of initial conditions tested.

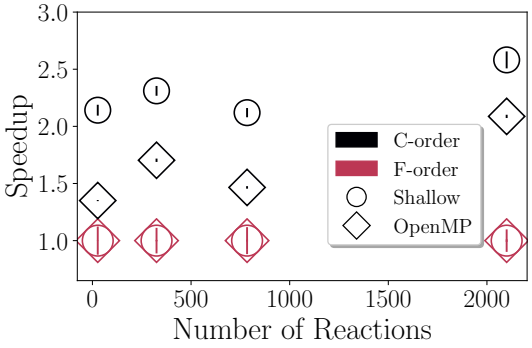



(b) The speedup achieved on the `K40m` versus the `C2075` GPU; the names correspond to the identifiers listed in Table 3.2

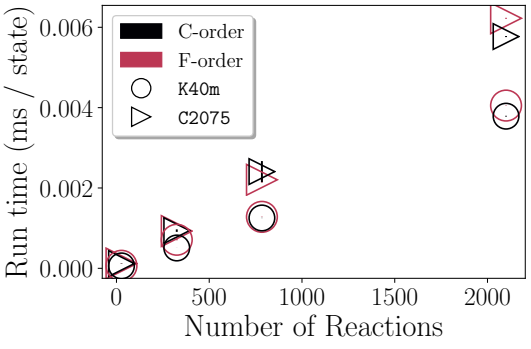
Figure 3.6: `pyJac` source-term evaluation performance on the Nvidia GPUs



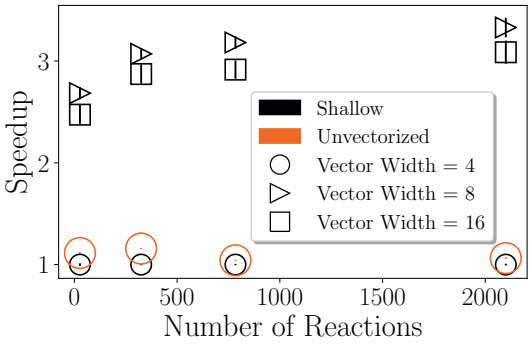

(a) The mean run time per condition for each chemical model using both the CONP and CONV formulations on Intel OpenCL and OpenMP on the `avx2` CPU.




(b) The speedup achieved by "C" ordering over "F" ordering for Intel OpenCL and OpenMP on the `avx2` CPU. The speedup presented is calculated per-language (OpenMP and OpenCL) to better assess the effect of the data ordering.




(c) The affect on performance of "C" vs "F"-ordering for the shallow-vectorized Nvidia OpenCL code on both GPUs with a vector-width of 128.




(d) The effect of vector-width on "C"-ordered shallow-vectorized Intel OpenCL source-term evaluations on the `avx2` CPU.

Figure 3.7: The effect of the CONP and CONV formulations, "C" and "F" data-ordering, and CPU vector-width on source-term evaluation performance in `pyJac`. The shallow-vectorized "C"-ordered OpenCL cases correspond to the vectorized data ordering described in Section 3.2.1.

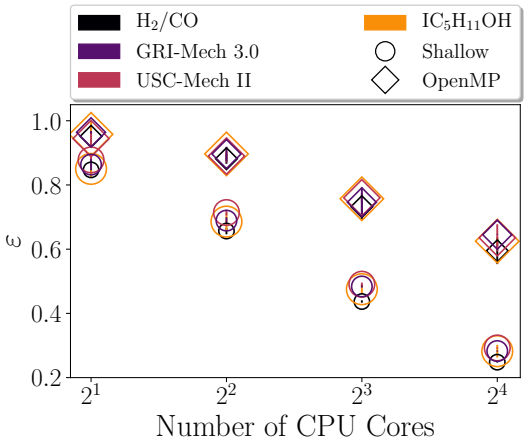
Additionally, the "C"-ordered shallow-vectorization in Fig. 3.7b and the other shallow-vectorized CPU data shown in Sections 3.3.5.1 and 3.3.5.2 use the vectorized-data ordering described in Section 3.2.1; this case achieves speedups of 2.14–2.58 × over the "F"-ordered shallow-vectorization, demonstrating the value of the vectorized-data ordering for CPU execution. Figure 3.7c shows how the "C"- and "F"-ordering affect the performance of source-term evaluation on both GPUs, with the speedup presented per-GPU. The "C"- and "F"-ordered shallow-vectorizations perform almost equivalently on both GPUs, with less than a 10 % difference in run time between data orderings. For the isopentanol model, "C"-ordered data is ~1.08 × faster on both GPUs (while the trend is less clear for the other models). The roughly equivalent performance between the "C" and "F"-ordered approaches on GPUs counters what one might expect: typically speaking, coalesced memory access in a shallow vectorization is easier to achieve with "F"-ordering (see Section 3.2.1). However, the vectorized-data ordering here ensures that memory storage is aligned to the vector width and, thus, encourages coalesced accesses. Figure 3.7d shows how changing vector width affects source-term evaluation performance on the `avx2` CPU. The vector width of 8 performs the fastest (out of those tested), while the larger vector width of 16 is slightly slower due to increased register pressure [141]. It is unclear why the vector width of 4 results in no speedup at all (in fact, it is the slowest case). Intel's vectorization guide [142] mentions that a heuristic determines the optimal vector width (in this case, it appears from compiler output to be 8), so it is possible that using a vector width smaller than the heuristic breaks the implicit vectorizer. This issue does not occur for a vector width of 4 on the `sse4.2` CPU.

Finally, Fig. 3.8 displays the (strong) parallel scaling efficiency and SIMD efficiency for the CPU platforms. The strong parallel scaling efficiency is defined as

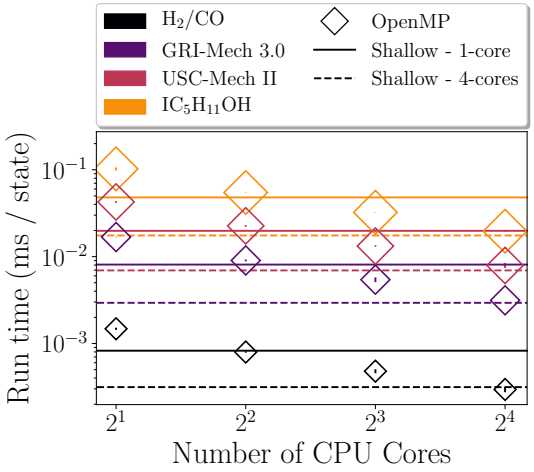
$$\varepsilon = \frac{\bar{t}_1}{N\bar{t}_N} , \tag{3.17}$$

where $\bar{t}_N$ is the mean run time per condition on $N$ CPU cores and $\bar{t}_1$ the same on a single CPU core. The strong parallel scaling efficiency measures the speedup due to the use of additional CPU cores as a fraction of linear speedup; strong scaling tends to decrease with the number of processors used due to memory-bandwidth limitations and decreasing computation work allocated per CPU core [143].

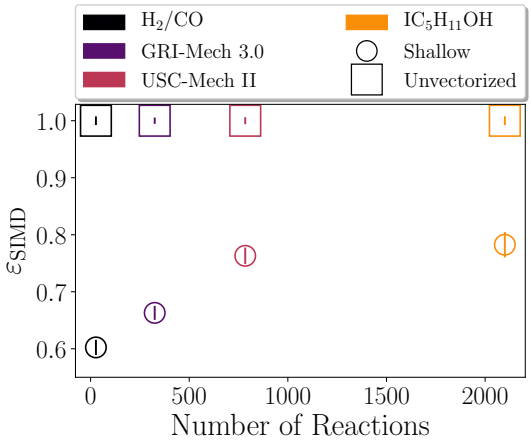
Figure 3.8a shows the strong parallel scaling efficiency of source-term evaluation in `pyJac` on the `avx2` machine for both the shallow-vectorized Intel OpenCL and OpenMP codes. In general, the $H_2$/CO mechanism has the worst scaling efficiency for both Intel OpenCL and OpenMP, likely resulting from both its relatively small size and few falloff/chemically activated reaction (in



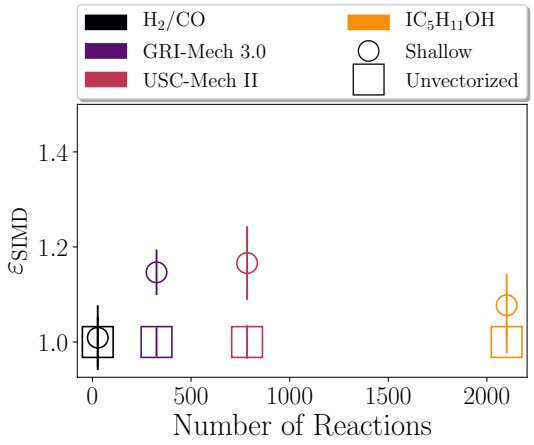

(a) The strong parallel scaling efficiency (as defined in Eq. (3.17)) of source-term evaluation for Intel OpenCL/OpenMP on the `avx2` machine.




(b) The mean run time per condition of OpenMP source-term evaluation on different cores compared to a shallow-vectorized Intel OpenCL evaluation on 1 (solid lines) and 4 (dashed lines) cores on `avx2` machine.




(c) The SIMD efficiency (Eq. (3.18)) of source-term evaluation for the Intel OpenCL runtime on a single core of the `avx2` CPU.




(d) The SIMD efficiency (Eq. (3.18)) of source-term evaluation for the Intel OpenCL runtime on a single core of the `sse4.2` CPU.

Figure 3.8: The parallel scaling efficiency and SIMD efficiency of source-term evaluation for Intel OpenCL on the `avx2` and `sse4.2` CPUs.

particular, the additional expensive logarithm and exponential evaluations that accompany them). As demonstrated in Appendix D, the amount of computational work required per thermochemical state plays a critical role in fully utilizing SIMD instructions/multiple threads. Additionally, OpenMP tends to scale better than the shallow-vectorized OpenCL code, e.g., ~0.9 and ~0.75 for four and eight CPU cores, respectively, compared to just 0.66–0.72 and 0.44–0.48 for OpenCL. Though not pictured (to keep the figure readable), the unvectorized Intel OpenCL code scales only slightly worse than the OpenMP code, hence the poorer scaling is unique to the shallow-vectorized code. This is due in large part to the superior performance of the shallow-vectorized OpenCL code, coupled with the relatively small amounts of work associated with source-term evaluation. To illustrate this, Fig. 3.8b shows the mean run time per-condition of the OpenMP source-term evaluations for 2–16 cores, compared to the shallow-vectorized OpenCL code on one (solid line) and four (dashed line) cores on the `avx2` machine. For all chemical models, the mean run time per-condition (and hence the computational work allocated per-core, one of the key-drivers of parallel scaling efficiency [143]) of OpenMP running on four cores is roughly equal to that of the shallow-vectorized OpenCL code on a single core. Similarly, OpenMP running on 16 cores is roughly equivalent to the OpenCL code on 4 cores. Therefore, a more fair comparison of parallel scaling efficiencies is to compare OpenCL running on 4 cores with OpenMP on 16; the OpenMP code's parallel efficiency drops to ~0.64 for 16 cores, similar to OpenCL's parallel scaling efficiency of 0.66–0.72 at 4 cores. Indeed, as will be seen in Section 3.3.5.2, sparse Jacobian evaluation—the most computationally intensive task in this work—exhibits similar strong-scaling efficiency on Intel OpenCL and OpenMP.

The SIMD efficiency is defined as

$$\varepsilon_{\text{SIMD}} = \frac{\bar{t}_{\text{unvec}}}{W\bar{t}_{\text{shallow}}}\,, \tag{3.18}$$

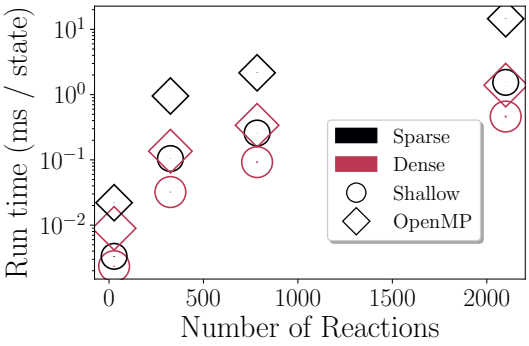
where $\bar{t}_{\text{unvec}}$ is the mean run time per condition of the unvectorized OpenCL code, $\bar{t}_{\text{shallow}}$ the same for the shallow-vectorized OpenCL code, and $W$ is the vector width reported in number of double operations (see Table 3.1). This measure compares the actual speedup due to shallow vectorization with the ideal speedup based on the nominal vector width of the machine.

Figure 3.8c shows the SIMD efficiency of source-term evaluation in `pyJac` on a single core of the `avx2` machine; the larger models (isopentanol and USC-Mech II) have higher SIMD efficiencies of 0.76–0.78, and the smaller models ($H_2$/CO, GRI-Mech 3.0) have lower SIMD efficiencies of 0.6–0.66. This again demonstrates that the SIMD vectorization becomes more efficient with increasing amounts of work to perform (i.e., with increasing model size). Interestingly, Fig. 3.8d shows the SIMD efficiency on the `sse4.2` machine as greater than one. This is likely caused by a combination of using an OpenCL vector width greater than the native CPU vector width (i.e., eight versus two) and improved data locality for the vectorized-data ordering as discussed

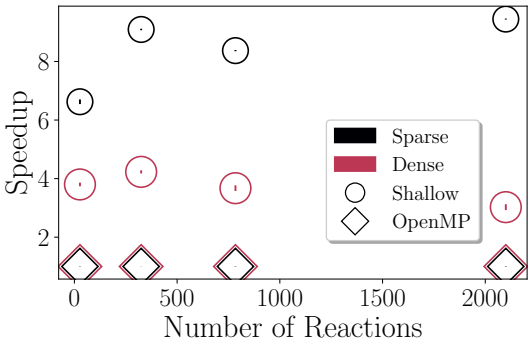
in Section 3.2.1, and results in a modest 7–17 % improvement over the nominal vector width.

#### 3.3.5.2 Jacobian evaluation

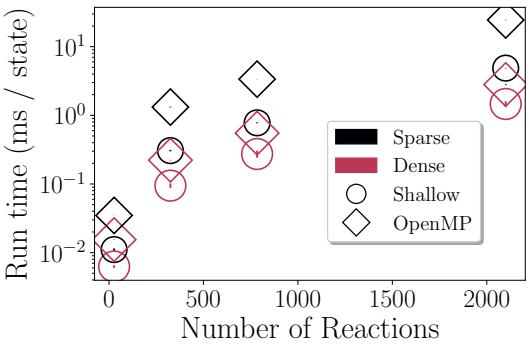
Figure 3.9 shows the performance of the sparse and dense Jacobian evaluations in `pyJac` on the CPU platforms. In Fig. 3.9a, the mean run time per condition is presented for the shallow-vectorized Intel OpenCL and OpenMP codes on the `avx2` CPU. The sparse Jacobian evaluates slower on both Intel OpenCL and OpenMP due to indirect lookup indexing requirements, as discussed in Section 3.3.4. Interestingly, indirect lookup less-negatively impacts the shallow-vectorized OpenCL code: the sparse OpenMP code is 2.47–10.42 × slower than the dense OpenMP evaluation, while the sparse shallow-vectorized OpenCL code is just 1.41–3.34 × slower than its dense counterpart. As a result, the shallow-vectorized sparse OpenCL code performs as fast or faster than the dense OpenMP code in all cases on the `avx2` machine (Fig. 3.9a). Figure 3.9b shows the speedup of the shallow-vectorized OpenCL, sparse and dense Jacobian evaluations over the same on OpenMP; the dense OpenCL code is 3.03–4.23 × faster than the corresponding dense OpenMP code. This speedup increases to 6.63–9.44 × for the sparse Jacobian.



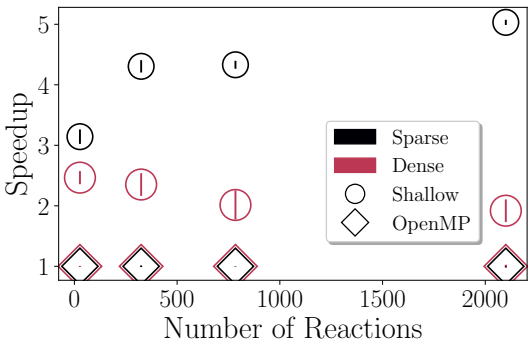

(a) The mean run time per condition of sparse and dense Jacobian evaluations for Intel OpenCL/ OpenMP on the `avx2` machine.




(b) The speedup of the sparse and dense Jacobian evaluations for Intel OpenCL over the sparse/dense OpenMP baseline on the `avx2` machine.



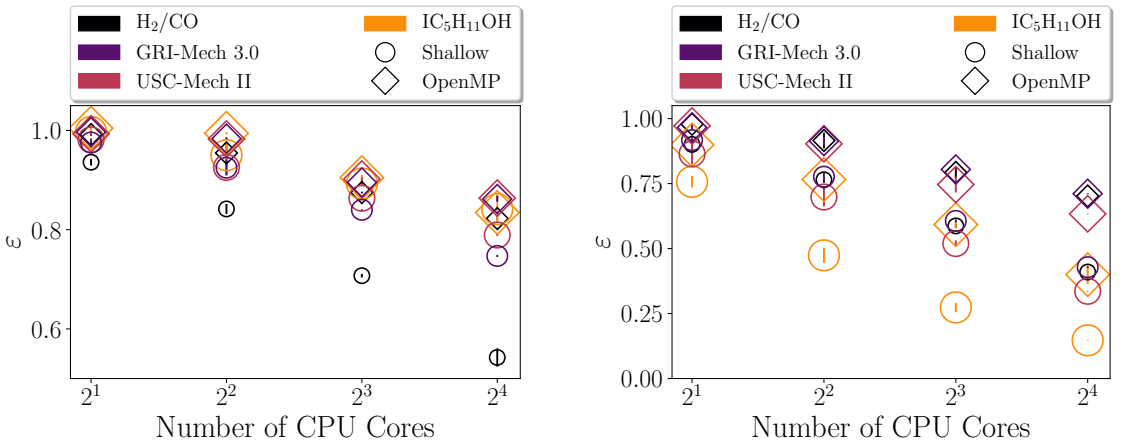

(c) The mean run time per condition of sparse and dense Jacobian evaluations for Intel OpenCL/ OpenMP on the `sse4.2` machine.




(d) The speedup of the sparse and dense Jacobian evaluations for Intel OpenCL over the sparse/dense OpenMP baseline on the `sse4.2` machine.

Figure 3.9: Performance of sparse and dense Jacobian evaluations on the CPU platforms in `pyJac`.

On the `sse4.2` machine, Fig. 3.9c shows similar results: the sparse OpenMP code is the slowest in all cases, and the shallow-vectorized OpenCL code is nearly as fast as the dense OpenMP code. Once again, indirect lookup less-negatively impacts the sparse OpenCL code, which is only 1.76–3.33 × slower than its dense counterpart, while the sparse OpenMP code is significantly (2.25–8.72 ×) slower than the dense version. In Fig. 3.9d, the speedup of the sparse and dense shallow-vectorized OpenCL codes are compared with their OpenMP versions; the dense OpenCL code is 1.92–2.47 × faster while the sparse shallow-vectorization achieves a speedup of 3.14–5.03 ×.




(a) Sparse Jacobian evaluation parallel scaling efficiency for Intel OpenCL/OpenMP on the `avx2` machine.

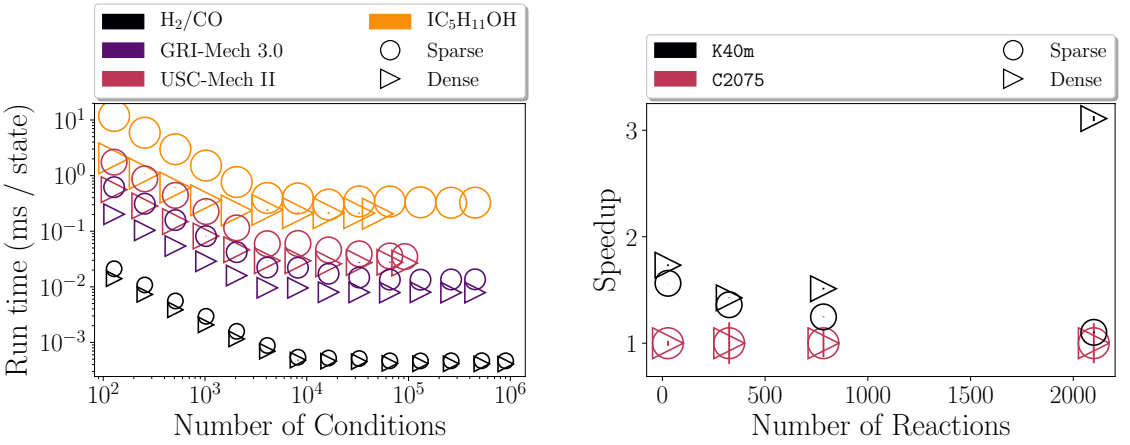
(b) Dense Jacobian evaluation parallel scaling efficiency for Intel OpenCL/OpenMP on the `avx2` machine.

Figure 3.10: Strong parallel scaling efficiencies of sparse and dense Jacobian evaluations for the shallow-vectorized Intel OpenCL and OpenMP codes on the `avx2` CPU.

Figure 3.10a compares the strong parallel scaling efficiency of the sparse shallow-vectorized OpenCL with the sparse OpenMP code. Although the plot is challenging to read since most of the data are clustered together, it shows that the shallow-vectorized OpenCL code scales similarly to the OpenMP code, in contrast to the parallel scaling efficiency of source-term evaluation (Fig. 3.8a). The $H_2$/CO model scales the worst for both codes, ranging from 0.94–0.54 and 0.99–0.82 efficiency for OpenCL and OpenMP respectively on 2–16 cores. As the model size increases, the efficiency of the OpenCL code improves dramatically, reaching 0.997–0.84 for the isopentanol model.

Figure 3.10b shows scaling for the OpenMP and shallow-vectorized dense Jacobian OpenCL codes. In this case, the isopentanol model scales the worst for both cases. The sheer size of the dense isopentanol Jacobian limited the total number of states for the dense isopentanol Jacobian evaluation to 50,000—storing the dense matrix for a single thermochemical state takes over 1 MB of data, so 50,000 states requires over 50 GB of memory—this greatly drops the computation cost for this case, and adversely affects the scaling efficiency as discussed in Section 3.3.5.1. Excluding

the isopentanol model, the dense shallow-vectorized OpenCL code scales slightly better than for source-term evaluation: 0.70–0.78 and 0.52–0.61 for four and eight cores, respectively (compared with 0.66–0.72 and 0.44–0.48 for shallow-vectorized OpenCL source-term evaluations). This results from the higher computational cost of Jacobian evaluation, and hence more available work per CPU core. As in Section 3.3.5.1, the shallow-vectorized dense Jacobian code running on 1 and 4 cores of the `avx2` machine performs roughly as fast as the OpenMP code on 4 and 16 cores, respectively. The parallel scaling efficiency of OpenMP on 16 cores (excluding isopentanol) is 0.63–0.71, similar to the shallow vectorization's efficiency of 0.70–0.78 for 4 cores.



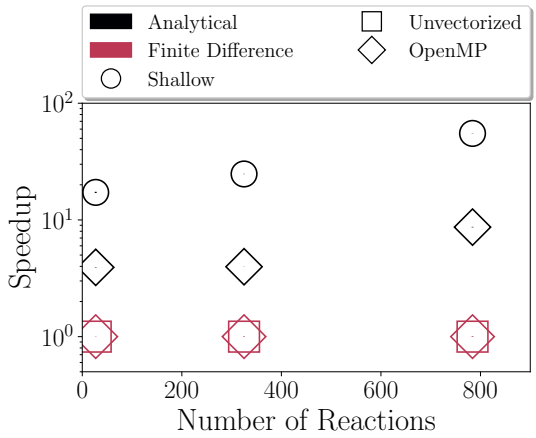

(a) Mean run time per condition of sparse and dense Jacobian evaluations on the `K40m` GPU.

(b) Speedup of the `K40m` over `C2075` GPUs for sparse and dense Jacobian evaluations, normalized per-Jacobian type (i.e., sparse versus dense).

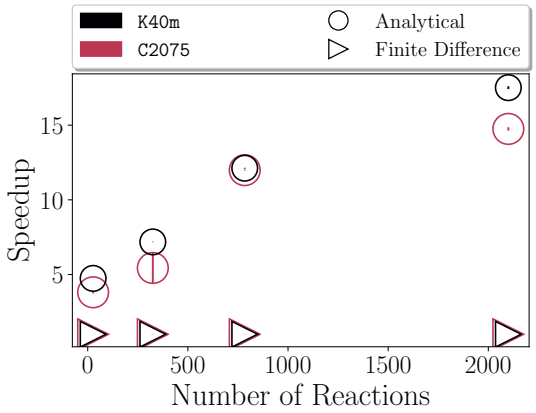
Figure 3.11: The performance of sparse/dense Jacobian evaluation on the `K40m` and `C2075` GPUs.

Figure 3.11a plots the mean run time per condition for the sparse and dense Jacobian evaluations on the `K40m` GPU. As in the species evaluation case, the mean run time per condition drops steadily, for both cases becoming roughly constant after just $3 \times 10^3$ states (except for $H_2$/CO, which levels off near $10^4$ conditions). Sparse Jacobian evaluation is significantly slower for all models before the GPU becomes saturated (due to the indirect indexing lookup), but the performance gap between sparse and dense evaluations narrows past the saturation point. This is likely due to the ability to fit many more sparse Jacobian matrices in the `K40m`'s memory, as well as improved data locality/caching due to the smaller size of the sparse Jacobian. Figure 3.11b presents the speedup of the `K40m` over the `C2075` GPU for sparse/dense Jacobian evaluation; sparse evaluation on the `K40m` is 1.10–1.59 $\times$ faster than on the `C2075`, while dense evaluation is 1.36–3.0 $\times$ faster. The speedup on the `K40m` decreases with increasing model size for the sparse formulation, but increases for larger models when dense; this likely results from the larger available memory of the `K40m`, and hence fewer data-transfer operations to/from the GPU. Figure 3.12 compares the performance of the sparse analytical Jacobian with a sparse first-order

finite-difference Jacobian on both the `avx2` CPU and `C2075`/`K40m` GPUs. Figure 3.12a shows large speedups for both OpenMP and shallow-vectorized OpenCL; the analytical OpenMP Jacobian is 3.92–8.67 × faster, while the analytical OpenCL Jacobian achieves speedups of 17.22–55.11 ×. We excluded the isopentanol case here, since a single run of the sparse finite-difference Jacobian using either OpenCL or OpenMP took over 12 hours of run time. In addition, the current finite-difference formulation breaks Intel's auto-vectorizer, and hence we compared OpenCL against the unvectorized OpenCL code (we did not prioritize fixing this issue, since we implemented the finite-difference Jacobian for comparison purposes only). Although we do not display the dense finite-difference Jacobian speedup in Fig. 3.12a, the dense OpenCL and OpenMP analytical Jacobian codes outperform the finite-difference variants by even larger margins: 24.44–245.63 × for OpenCL and 9.68–112.73 × for OpenMP (these data do include the isopentanol model, though limited to 50,000 conditions as discussed previously). Figure 3.12b compares the sparse analytical and finite-difference Jacobians on the GPUs. The analytical Jacobian on the `K40m` and `C2075` shows speedups of 3.81–17.60 × that increase with chemical model size; the `K40m` has a larger speedup than the `C2075` for the isopentanol model (17.60 × vs. 14.75 ×) due to its larger available memory. Although not pictured, the dense analytical Jacobian on the `K40m` GPU has larger speedups compared with the dense finite-difference Jacobian: 4.04–45.13 ×. The `K40m` GPU again shows significantly larger speedups over the `C2075` for the isopentanol model (45.13 × vs. 23.85 ×), further underscoring the effect of more available memory on the `K40m`.




(a) Speedup of the sparse analytical Jacobian versus finite-difference Jacobian evaluation on the `avx2` CPU, normalized per-language.

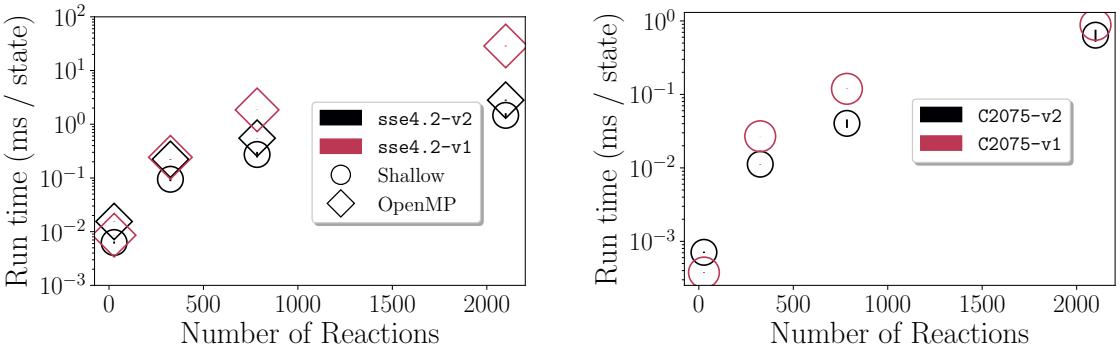



(b) Speedup of the analytical versus finite-difference Jacobian evaluation on both the `K40m` and `C2075` GPUs, normalized per-GPU.

Figure 3.12: The performance of a sparse, first-order forward finite-difference Jacobian compared to the analytical Jacobian on the `avx2` CPU and both GPUs.

Figure 3.13 compares the performance of evaluating the dense analytical Jacobian of `pyJac-v2` with that of the previous version, `pyJac-v1` [124], on the `sse4.2` CPU and `C2075` GPU. (We

selected dense Jacobian evaluation for this comparison since it was the only type implemented in the previous version of `pyJac`.) On the `sse4.2` CPU, the `pyJac-v2` evaluates faster than `pyJac-v1` for OpenMP for the larger chemical models; the static OpenMP code generated by `pyJac-v1` (see Section 3.2.3) is 1.79 × faster for the $H_2$/CO model, and only 1.09 × slower for the GRI-Mech 3.0 model. In contrast, the loop-based OpenMP code of `pyJac-v2` is 3.37–10.19 × faster than `pyJac-v1` for the USC-Mech II and isopentanol models. The shallow-vectorized OpenCL `pyJac-v2` code is faster than `pyJac-v1` in all cases, achieving speedups of 1.37–19.56 × that increase with model size. Figure 3.13b compares the performance of the `pyJac-v2` with `pyJac-v1` for evaluating dense analytical Jacobians on the `C2075` GPU. As with the CPU, the static code of `pyJac-v1` is slightly faster for the $H_2$/CO model, but `pyJac-v2` outperforms `pyJac-v1` by 1.25–2.84 × for the other models. Performance likely drops for the isopentanol model due to the limited number of conditions in dense evaluation for `pyJac-v2`, as noted earlier; the speedup of `pyJac-v2` relative to `pyJac-v1` is expected to be greater than shown in Fig. 3.13 if the full set of conditions were used.




(a) `sse4.2` CPU (b) `C2075` GPU

Figure 3.13: Performance comparison of dense Jacobian evaluation using `pyJac-v2` and `pyJac-v1` [124].

## 3.4 Practical notes on OpenCL use

While OpenCL provides a simple interface to enable cross-platform execution, and significant speedups were achieved via shallow-vectorized OpenCL code in this work, there are some serious potential pitfalls in its use. The closed-source OpenCL runtimes tested in this work (Intel and Nvidia) contain several bugs that result in compilation failures, simply incorrect vectorized machine code, or even segmentation faults. Further, these runtimes (in our experience) tend to be less responsive to fixing said bugs, with relatively infrequent new releases (or in Nvidia's case, lack of changelogs/public records of bug-fixes). On the other hand, the open-source OpenCL runtime used in this work, POCL, has far fewer implementation bugs, and when issues arise the

community is very responsive to bug reports and user outreach; however, POCL fails to achieve vectorization as noted in Section 3.3.5.1.

To demonstrate the type of issue discussed, we created a minimum working example [144] that shows a failure of Nvidia's OpenCL runtime corresponding to the GPU driver version 375.26 on a Tesla `K40m` GPU; simply changing to another runtime (e.g., Intel) produces the correct result—with no code changes or recompilation. This provides a particularly vexing problem for the programmer, since little can often be done to resolve the issue; thankfully, in this case we were able to upgrade the driver version to resolve the problem. Further, as noted throughout this work, certain code-generation patterns can break OpenCL execution/vectorization, e.g., the failure of POCL to achieve vectorization or Intel OpenCL's failure to vectorize the finite-difference Jacobian. Indeed the vectorization process attempted by most OpenCL runtimes (and thus, reasons for incorrect/unvectorized code output) is obscured from the user, making reasoning about errors or performance trends quite difficult. Thankfully, `loo.py` allows relatively easy switching between output languages; the most significant code change requires building of the wrapper that initializes/transfers memory and calls the source-rate/Jacobian kernel. These issues make it critical to provide adequate implementation details (e.g., runtime version, platform, etc., as given in Section 3.3.1) for codes utilizing OpenCL in order to enable reproducible results.

## 3.5 Conclusions

In this work, we developed automatically generated OpenCL codes for SIMD and SIMT-vectorized evaluations of thermochemical source-term and sparse/dense chemical kinetic Jacobian matrices. The main contributions of this work are:

- Deriving and verifying a new Jacobian formulation that greatly increases sparsity;
- Enabling vectorized execution on the CPU, GPU, and other accelerators; and
- Achieving significant speedups over a strictly parallel Jacobian and source-term evaluation on SIMD-enabled processors (e.g., CPUs).

These efforts are made publicly available (see Appendix D) via the open-source, high-performance, chemical kinetics code `pyJac`. The new molar-based formulation resulted in highly sparse chemical kinetic Jacobians, and allows selection of either the constant-volume or constant-pressure approximation. In addition, sparsity can be increased further by eliminating components associated with the bath gas, as discussed in Section 3.3.4; this approximation is not a key feature of this work, and future efforts to incorporate more sophisticated Jacobian approximations would be a worthwhile endeavor.

We also demonstrated source-term and Jacobian evaluation for a range of chemical kinetic models [88, 89, 135, 136] and multiple CPUs/GPUs (Tables 3.1 and 3.2). In addition to parallel OpenMP evaluation on the CPU, this work enabled the shallow-vectorized evaluation of the chemical-kinetic source terms and analytical Jacobian on both the CPU and GPU via OpenCL. Deep vectorization is possible on the Portable OpenCL (POCL) platform [129], but it yields no performance benefit as POCL did not achieve vectorizations for any execution pattern studied. Deep vectorization deserves further study with other platforms (e.g., CUDA).

We demonstrated significant speedups in shallow SIMD-vectorized execution over a parallel OpenMP code for evaluating the chemical-kinetic source terms and sparse/dense Jacobian: up to 4.09 ×, 9.44 ×, and 4.23 ×, respectively, on an `avx2`-capable CPU. Sparse Jacobians evaluate more slowly than dense Jacobians on all CPU/GPU platforms due to indirect lookup requirements in array indexing, but this adversely affects the shallow-vectorized OpenCL code less than OpenMP. Further, analytical Jacobians evaluate significantly faster than a first-order finite-difference-based approach on all platforms. Finally, we compared the performance of evaluating dense, analytical Jacobians in this new version of `pyJac` with the previous version [124]. The OpenMP version evaluates moderately slower on the CPU for the smallest chemical model (e.g., 1.79 × on the `avx2` CPU), but significantly faster for the larger models—up to 10.19 ×. The shallow-vectorized OpenCL code runs faster than the previous version over all chemical models, reaching speedups of 19.56 ×.

The OpenMP code-generation is currently only capable of parallel execution, but extending this platform to shallow/deep-vectorizations (via `loo.py` and compiler `#pragma`s) is a key priority going forwards since OpenMP is a standard library on most machines. In addition, CUDA [121] has been significantly more reliable in previous works [44, 110], while Intel's open-source OpenCL alternative, ISPC [145], has been relatively stable and easy to work with during preliminary efforts with the unit-testing discussed in Section 3.2.3. The current deep-vectorization formulation would be executable for both CUDA and ISPC targets, as these languages implement double-precision atomic operations, further recommending their use. It is also possible, particularly for the Intel OpenCL/POCL runtimes, that better performance/stability might be achieved using so-called "explicit" vectorization, i.e., through use of built-in vector types such as the `double8`. Specifically, this change could enable vectorization on the POCL runtime and might also enable deep vectorization on the Intel OpenCL runtime. Using OpenCL Image/CUDA Texture memory to accelerate the indirect lookup for sparse matrix evaluation should be investigated as well. Finally, future extensions of this work will include extending to additional target languages (e.g., vectorized OpenMP, CUDA, ISPC) to improve ease of use and reliability, improving the existing OpenCL targets (e.g., to enable meaningful deep-vectorized evaluation),

and implementing reaction sorting [53] to improve SIMD efficiency (Appendix D.3).

One key component missing in this work is vectorized sparse/dense linear-algebra subroutines to maximize the performance of LU-factorization and matrix-vector multiplication (commonly used in implicit-integration techniques). Third-party/open-source options exist for some target languages, e.g., cuBLAS [146], clBLAS/clSPARSE [147] or SuperLU [148], but these do not necessarily cover all targets/required linear-algebra operations and, in the case of the CUDA/OpenCL libraries, are often optimized operations on one large matrix instead of many (relatively) smaller matrices. The extent to which these existing programs may be used needs to be assessed and missing operations should be implemented in `loo.py` to ensure easy switching between target languages and vectorization types.

# Chapter 4

# Vectorized chemical kinetic integration in realistic reactive-flow simulations

## 4.1 Introduction

As single-instruction multiple-thread (SIMT) processors, e.g., graphics processing units (GPUs) have become more widely available, many studies have leveraged their high floating-operation throughput to accelerate chemical kinetic integration. Early studies [47, 48] focused on using the GPU to evaluate chemical kinetic source-terms, or factorize the Jacobian matrix, but found significant speedups only for large chemical kinetic models. Later, several works [49, 50, 52] implemented GPU-based explicit integration techniques to achieve order of magnitude or more speedups in the integration of non-stiff or moderately-stiff chemical kinetics. Concurrently, GPU-based implicit integration techniques were developed [51, 53, 54], but experienced performance degradation with increasing numerical stiffness due to thread-divergence concerns. In contrast, single-instruction multiple-data (SIMD) based vectorization, e.g., as found on a central processing unit (CPU), has not been as extensively investigated for use in chemical kinetic integration. Recently, Stone et al. [57] utilized the OpenCL [108] framework to vectorize a linearly-implicit Rosenbrock integration method [67] on the CPU, GPU and the Intel Many Integrated Core processor (MIC). In addition, Curtis et al. [149] have developed an open-source platform, `pyJac`—as described in Chapter 3—that can generate vectorized-OpenCL chemical kinetic source-term and Jacobian evaluation codes. This effort of this chapter aims to combine these two techniques to demonstrate the significant speedups that can be achieved by

fully-vectorized chemical kinetic integration on the CPU.

This work describes a methodology to implement SIMD-based vectorized ODE integration methods—coupled with vectorized analytical chemical kinetic Jacobian and source-rate evaluations—for use in a realistic reactive-flow simulation. In Section 4.2, the software, numerical methods and models used in this study will be outlined. Subsequently, in Section 4.3 the vectorized solvers will first be validated against a standard implicit integration algorithm [42, 150], after which their coupling to the computational fluid dynamics (CFD) code `OpenFOAM` [151] will be verified. Additionally, a case-study modeling the Sandia Flame D [152–154] will be detailed in Section 4.4, and used to demonstrate the precision of the solvers on a realistic reactive-flow simulation. Furthermore, Section 4.4 will explore the performance of the vectorized solvers as compared to those built into `OpenFOAM`. Finally, directions for future efforts will be identified in Section 4.5

## 4.2 Numerical methods and software

### 4.2.1 The `pyJac` code-generation platform

`pyJac` [149] is an open-source platform capable of generating code for the evaluation of the chemical kinetic source-terms and analytical Jacobian for a variety of execution, data-ordering, and matrix-format patterns. For full details, we direct the reader to Chapter 3, however here we will highlight key points relevant to this work. When using a constant-pressure assumption*, the resulting thermochemical state vector in `pyJac` is:

$$\Phi = \left\{T, V, n_1, n_2 \dots n_{N_{\text{sp}}-1}\right\} \tag{4.1}$$

where $T$ is the temperature of the gas in K, $V$ the volume in $\text{m}^3$, $n_i$ the moles of species $i$ in kmol, and $N_{\text{sp}}$ the number of species in the chemical kinetic model. The last species in the model is typically taken to be the bath-gas—$N_2$ in this study—and is omitted from the state-vector, as it is calculated implicitly via the ideal gas equation:

$$n = \frac{PV}{\mathcal{R}T} = \sum_{i=1}^{N_{\text{sp}}} n_i \ , \tag{4.2}$$

where $n$ is the total number of moles of the gas and $\mathcal{R}$ is the universal gas constant, and the following equation:

$$n_{N_{\text{sp}}} = \frac{PV}{\mathcal{R}T} - \sum_{i=1}^{N_{\text{sp}}-1} n_i \ . \tag{4.3}$$

*Note: in this context, "constant-pressure" refers to the solution of chemical kinetics within a reaction sub-step of the operator splitting scheme, rather than a general constant-pressure reactive-flow simulation.

This formulation results in an explicit conservation of mass in `pyJac`, as well as ensuring the system of equations is not over-constrained [116].

Given a thermochemical state vector, `pyJac` can evaluate both the chemical kinetic source rates:

$$\frac{\mathrm{d}\Phi}{\mathrm{d}t} = f\left(\Phi\right) = \left\{\frac{\mathrm{d}T}{\mathrm{d}t}, \frac{\mathrm{d}V}{\mathrm{d}t}, \frac{\mathrm{d}n_1}{\mathrm{d}t}, \frac{\mathrm{d}n_2}{\mathrm{d}t} \ldots \frac{\mathrm{d}n_{N_{\mathrm{sp}}-1}}{\mathrm{d}t}\right\} , \tag{4.4}$$

and the analytical chemical kinetic Jacobian:

$$\mathcal{J}_{i,j} = \frac{\partial f_i}{\partial \Phi_j}, \qquad i, j = 1 \ldots N_{\mathrm{sp}} + 1 \, . \tag{4.5}$$

Although `pyJac` can evaluate the Jacobian in a sparse-matrix format (e.g., compressed row storage), a dense-matrix format was used in this work to simplify the implementation of linear-algebra operations; extension of this effort to utilize a sparse-matrix is a goal for future studies. For this study, `pyJac` was used to generate an explicit "wide"-vectorization (as will be described in Section 4.2.2) using a vector-width of 8 double-precision floating point operations*, and "C" (or row-major) data-ordering.

### 4.2.2 OpenCL and vectorization

The parallel programming standard OpenCL [108] provides a common interface to execute vectorized code on a variety of different platforms, e.g., the CPU, GPU and MIC. For a detailed overview of different vectorization patterns and their applications for different hardware platforms in the integration of chemical kinetic ODEs, we refer the reader to past-works [51, 54, 117, 149]. Here we will discuss only the so-called "wide"-vectorization pattern for SIMD-processors (e.g., CPUs) using OpenCL.

In this method, the thermochemical state-vector of several chemical kinetic ODEs are grouped together, e.g.:

$$\Phi_{\mathrm{vec}} = \left\{\left\{\Phi_{1,1}, \Phi_{2,1} \ldots, \Phi_{N_v,1}\right\}, \left\{\Phi_{1,2}, \Phi_{2,2} \ldots, \Phi_{N_v,2}\right\} \ldots, \left\{\Phi_{1,N_{\mathrm{sp}}+1}, \Phi_{2,N_{\mathrm{sp}}+1} \ldots, \Phi_{N_v,N_{\mathrm{sp}}+1}\right\}\right\} \tag{4.6}$$

where $N_v$ is the number of elements in the vector, also known as the "vector-width", and $\Phi_{j,i}$ is the $i$th component of the thermochemical state-vector (Eq. (4.1)) for the $j$th state.

These modified state-vectors can be loaded into OpenCL vector data-types, e.g., `double8`, allowing floating point math operations to be performed concurrently (by specialized vector-processors present on all modern CPUs) over the vector, resulting in accelerated

*We note that if the OpenCL vector-width is longer than that implemented on the underlying hardware, the vector-operation is implicitly converted to multiple smaller vector-operations, similar to loop-unrolling optimizations.

computations. The OpenCL runtime (e.g., as supplied by Intel [132]) is then responsible for transforming the OpenCL code into vectorized operations using the vector-instruction set present on the device.

### 4.2.3 The `accelerInt` ODE integration package

The efficient and accurate integration of the chemical kinetic ODEs is a critical to the accuracy and performance of reactive-flow simulations. During integration, the source-rates—detailed in Section 4.2.1—are advanced from initial time $t_i$ to a final time $t_f$:

$$\frac{\mathrm{d}\Phi}{\mathrm{d}t} = f\left(\Phi\right), \quad t_i \leq t \leq t_f \; . \tag{4.7}$$

As a part of this effort, several previously developed vectorized-OpenCL ODE integration methods [57] have been incorporated into the `accelerInt` [54] software package. These include a fourth-order explicit Runge-Kutta method as well as several third and fourth order linearly-implicit Rosenbrock methods [67, 155, 156]. Table 4.1 lists the OpenCL solvers available in `accelerInt`, as well as their order, solver-type and references.

| Solver Name | Order | Solver Type | Reference(s) | Short-name |
|---|---|---|---|---|
| Rosenbrock | 3 | Linearly-implicit | [155, 156] | `ROS3` |
| Rosenbrock | 4 | Linearly-implicit | [67] | `ROS4` |
| RODAS | 3 | Linearly-implicit | [155, 156] | `RODAS3` |
| RODAS | 4 | Linearly-implicit | [67] | `RODAS4` |
| RKF45 | 4 | Explicit | [157] | `RKF45` |

Table 4.1: Listing of vectorized OpenCL integration methods incorporated into `accelerInt` as a part of this effort.

Runge–Kutta (RK) methods include both implicit and explicit solvers, and are widely used to solve systems of both stiff and non-stiff ODEs. An RK method with $s$ stages may be written as:

$$\Phi\left(t_{n+1}\right) = \Phi\left(t_n\right) + \sum_{i=1}^{s} b_i \boldsymbol{k}_i \tag{4.8}$$

where each stage is computed as:

$$\boldsymbol{k}_i = hf\left(\Phi\left(t_n\right) + \sum_{j=1}^{s} a_{ij}\boldsymbol{k}_j\right) \tag{4.9}$$

Here, $h$ is the adaptive integration time-step, and $a_{ij}$ and $b_i$ are method coefficients that define the algorithm.

The explicit solver included in `accelerInt` is a five-stage, fourth-order accurate embedded Runge-Kutta-Fehlberg method [157]. Explicit RK methods are obtained when the coefficient

matrix $a_{ij}$ in Eq. (4.9) is strictly lower-triangular (i.e., $a_{ii} = 0$). While explicit integration methods are efficient for non-stiff problems, they are only conditionally stable and perform poorly for stiff problems where the step-size $h$ is limited by stability concerns rather than the desired accuracy.

Implicit RK integration methods result from a fully populated coefficient matrix $a_{ij}$, resulting in a system of non-linear equations that are typically solved via Newton–Raphson iteration. This technique requires the repeated evaluation and factorization of the chemical kinetic Jacobian $\mathcal{J}$ (see Eq. (4.5)). As such, the cost per integration-step is higher for implicit methods, however the improved stability achieved with such techniques typically makes them more efficient for the solution of stiff ODEs. Implicit methods commonly reuse the Jacobian matrix (and factorization) for multiple time-steps to reduce the computational overhead.

Rosenbrock-methods (ROS)* are more similar in structure to RK methods than to fully implicit techniques, solving a linearized form of Eq. (4.8) and are therefore known as "linearly-implicit" techniques. An $s$-stage ROS method is formulated as:

$$\boldsymbol{k}_i = hf\left(\Phi\left(t_n\right) + \sum_{j=1}^{i-1} \alpha_{ij}\boldsymbol{k}_j\right) + h\mathcal{J}\sum_{j=1}^{i} \gamma_{ij}\boldsymbol{k}_j, \quad i = 1, \ldots, s \tag{4.10}$$

$$\Phi\left(t_{n+1}\right) = \Phi\left(t_n\right) + \sum_{i=1}^{s} b_i\boldsymbol{k}_i \tag{4.11}$$

where $\alpha_{ij}$ and $\gamma_{ij}$ are the method parameters. Typically ROS methods are constructed with $\gamma_{ij} = \gamma\,\forall i$, a constant parameters, and $\alpha_{ij}$ as a lower triangular matrix, such that the stages may be solved sequentially and only a single LU-decomposition must be performed per time-step.

To avoid the solution of a linear system and matrix-vector multiplication at each stage of the method [67, 158], Eq. (4.10) may be transformed:

$$\left(\frac{1}{h\gamma_{ii}}\boldsymbol{I} - \mathcal{J}\right)\boldsymbol{u}_i = f\left(\Phi\left(t_n\right) + \sum_{j=1}^{i-1} a_{ij}\boldsymbol{u}_j\right) + \sum_{j=1}^{i} \frac{c_{ij}}{h}\boldsymbol{u}_j, \quad i = 1, \ldots, s \tag{4.12}$$

$$\Phi\left(t_{n+1}\right) = \Phi\left(t_n\right) + \sum_{j=1}^{s} m_j\boldsymbol{u}_j \tag{4.13}$$

where:

$$\Gamma = \gamma_{ij} \tag{4.14}$$

*Here we adopt the naming convention of Harier and Wanner [67].

is an intermediate value taken to be the matrix of $\gamma_{ij}$ values, and:

$$\boldsymbol{u}_i = \sum_{j=1}^{i} \gamma_{ij} \boldsymbol{k}_j \ , \tag{4.15}$$

$$a_{ij} = \alpha_{ij} \Gamma^{-1} \ , \tag{4.16}$$

$$c_{ij} = \gamma \boldsymbol{I} - \Gamma^{-1} \ , \tag{4.17}$$

$$m_j = b_j \Gamma^{-1} \ . \tag{4.18}$$

The direct use of the Jacobian matrix in ROS-solvers avoids the need for Newton-iteration making these methods particularly well-suited for SIMD and SIMT-vectorization due to the low-levels of divergence between vector-lanes/threads. However, ROS-solvers are formulated around the use of an exact (analytical) Jacobian, as the use of finite-difference Jacobian may impact the solver's order-conditions and convergence [67, 159]. Further, the Jacobian must now be evaluated/factorized at each time-step adding to the cost per time-step; W-methods, formulated around the use of an inexact Jacobian [67], may be a suitable technique to avoid these costs, and should be investigated for vectorized ODE-integration in the future.
Coupled with OpenCL source-rate/Jacobian evaluation code generated by `pyJac` [149], these solvers form the basis of the accelerated ODE-integration techniques used in this work.

### 4.2.4 `OpenFOAM`

`OpenFOAM` [151] is an open-source C++ library capable of solving a variety of continuum mechanics problems, and is designed to allow easy extension and implementation of custom solvers. In this work, the `OpenFOAM` applications targeting the solution of reactive-flow CFD problems will be extended to incorporate the vectorized ODE integration techniques outlined in Section 4.2.3. The reactive-flow solver `reactingFoam` is capable of utilizing a variety of turbulence models, including Reynolds-averaged Navier-Stokes (RANS) and large-eddy simulations (LES), boundary conditions, models and solution techniques. In this study we have exclusively used the standard k-$\varepsilon$ RANS turbulence model [160, 161] in `OpenFOAM`, however future efforts will extend this work to high-resolution LES studies. For comprehensive details on these models, and their implementation in OpenFOAM we refer the reader to e.g., [151, 162–164]. Here we will focus upon the models relevant to the incorporation of `accelerInt` into `OpenFOAM` for the solution of chemical kinetic ODEs.

#### 4.2.4.1 Chemistry Solvers

OpenFOAM has a number of built-in solvers for integration of the chemical kinetic ODEs, including C++ implementations of each of the third- and fourth-order linearly-implicit Rosenbrock and RODAS solvers found in accelerInt. However, the implementations differ in a number of key aspects. First, and most obviously, the difference in programming languages results in different execution patterns, i.e., serial evaluation in OpenFOAM vs. vectorized/batched integration in accelerInt. Secondly, the ROS4 solver in OpenFOAM uses a different set of method coefficients [159], which are claimed to be more-accurate at small step-sizes, but less stable than those [67] used in accelerInt for large step-sizes. Third, while (as of June 2018) OpenFOAM has an analytical Jacobian evaluation code built-in*, it utilizes a different state-vector composed of the species-concentrations, temperature, and pressure (which is assumed to be constant, as in pyJac). Finally, OpenFOAM does not employ the explicit mass-conservation formulation employed by pyJac, that is, the concentration of the last species in the model is solved for directly in OpenFOAM.

#### 4.2.4.2 Batched chemical kinetic integration

A new chemistry model, named the BatchedChemistryModel, was created by extending the base OpenFOAM class BasicChemistryModel. The key difference in this new model is that the chemical kinetic integration of the thermochemical state vectors for the domain (or sub-domain, in the case of runs parallelized with the message passing interface, MPI [165]) is performed in a single batched call to the accelerInt library, instead of being evaluated sequentially (as in the base OpenFOAM code). The accelerInt library returns the updated thermochemical state vectors for the domain, as well as the final internal integration time-step taken for each cell in the domain. The time-step values are used both as an initial time-step for the next ODE integration of this cell (as in the base OpenFOAM code), as well as to adaptively limit the overall CFD time-step for certain OpenFOAM solvers (e.g., chemFoam).

#### 4.2.4.3 Turbulent combustion model

OpenFOAM contains implementations of a number of turbulent combustion models used to simulate turbulence-chemistry interactions in reactive-flow simulations, ranging from a simple infinitely-fast chemistry model, to the more complex flame surface density [166] formulation. For our purposes we have selected the eddy-dissipation concept (EDC) [167] as a robust, and relatively computationally intensive combustion model to demonstrate the performance of the

*Previous versions of OpenFOAM used a semi-analytical approach, where most Jacobian values were computed analytically, but the temperature derivatives were evaluated using finite-differences.

vectorized ODE solvers. EDC is a commonly-used technique for the modeling of turbulence-chemistry interaction, and has been applied to a large variety of combustion problems e.g., [168–170]. Conceptually, EDC is based upon the idea of turbulence energy cascade [167] and involves both a fine-scale reactor and its surroundings; reactions may occur in either, however the surroundings are typically treated as non-reacting [168]. Molecular mixing between the fine-scale and the surroundings is modeled by mass-transfer between the two. When using a detailed chemical kinetic model, the fine-scale reactor is typically treated as a perfectly-stirred reactor and solved to equilibrium [171], imposing significant computational overhead.

The mean reaction rate for species $i$ in the EDC model [167, 168], $\overline{R}_i$, is given by*:

$$\overline{R}_i = \frac{\overline{\rho}}{\tau^*} \frac{\gamma_L^2 \chi}{1 - \gamma_L^2 \chi} \left( \overline{Y}_i - Y_i^* \right) \tag{4.19}$$

where $\overline{\rho}$ is the mean fluid density, $\overline{Y}_i$ and $Y_i^*$ the fluid mean and fine-structure mass-fractions of species $i$, respectively, $\tau^*$ and $\gamma_L$ the fine-structure residence time and dimensionless length fraction, respectively, and $\chi$ the fraction of fine-structure regions that interact with the rest of the fluid, typically assumed to be unity [168].

By definition, we have:

$$[C_i] = \rho \frac{Y_i}{W_i} = \frac{n_i}{V} \tag{4.20}$$

where $W_i$ and $[C_i]$ are the molecular weight and concentration of species $i$, respectively while the volume in Eq. (4.20) is taken to be that of the corresponding CFD-cell. Combining with Eq. (4.19) gives:

$$\overline{R}_i = \frac{1}{\tau^*} \frac{\gamma_L^2 \chi}{1 - \gamma_L^2 \chi} W_i \left( [\overline{C}_i] - [C_i^\diamond] \right) \tag{4.21}$$

where $[C_i^\diamond]$ is defined as:

$$[C_i^\diamond] = \overline{\rho} \frac{Y_i^*}{W_i} = \frac{n_i^*}{\overline{V}} \tag{4.22}$$

Using Eqs. (4.20) and (4.22), Eq. (4.21) may be rewritten in terms of species moles (the species state-variable used in `pyJac`) as:

$$\overline{R}_i = \frac{1}{\tau^*} \frac{\gamma_L^2 \chi}{1 - \gamma_L^2 \chi} \frac{W_i}{\overline{V}} \left( \overline{n}_i - n_i^* \right) \tag{4.23}$$

Finally, we note that both $\overline{n}_{N_{\text{sp}}}$ and $n^*_{N_{\text{sp}}}$, the number of moles of the last species in the model in the mean and fine-structures, respectively, are calculated using Eq. (4.3) to be consistent with `pyJac`.

*For the sake of this derivation, we only consider Magnussen's 2005 EDC model [167]. `OpenFOAM` implements other versions of the EDC model (see [168] for a good overview of the available versions), however the version selected does not affect the derivation of $\overline{R}_i$, as the chemical kinetic model is only responsible for evaluating the thermodynamic components of Eq. (4.23).

## 4.3 Validation

### 4.3.1 `accelerInt` Validation

To validate the new solvers, 100,000 thermochemical conditions were sampled from a previously generated database [172], created using a constant-pressure partially stirred reactor simulation [44] with the GRI-Mech 3.0 [89] chemical kinetic model. The database contains pressures ranging from 1–25 atm and covers a range of temperatures and compositions from cold unburned methane-air mixture to ignition and equilibrium. These conditions were integrated using `CVODEs` [42, 150] with very tight integration tolerances (ATOL $= 10^{-20}$, RTOL $= 10^{-15}$) for a single global integration time-step of $\Delta t = 10^{-6}$ sec to form a reference solution.
The OpenCL solvers were then used to integrate the same initial values to the same end-time, varying the tolerances given to the adaptive time-stepping algorithm such that:

$$\text{TOL} = 10^{-4}, 10^{-5} \ldots, 10^{-15}$$

We note that both the relative and absolute tolerances for the OpenCL solvers were set to TOL for this validation effort; in general these can be (and often are) different, e.g., as in the computation of the reference `CVODEs` solution.
The error of each initial value problem (IVP) was then measured as:

$$\|E_j\| = \left\| \frac{|y^{\circ}_{i,j}(t) - y_{i,j}(t)|}{10^{-10} + 10^{-6} \times |y^{\circ}_{i,j}(t)|} \right\|_2 , \tag{4.24}$$

where $y^{\circ}_{i,j}(t)$ is the $i$-th component of the solution computed by `CVODEs` for the $j$-th IVP and $y_{i,j}(t)$ the solution computed by the solver being tested. The "tolerances" used for calculation of the weighted reference solution components in Eq. (4.24) are for normalization purposes only.
The error over all IVPs was then calculated using the infinity-norm:

$$\|E\| = \|E_j\|_{\infty} . \tag{4.25}$$

The tested solvers used the same vectorization settings outlined in Section 4.2.1. Figure 4.1 shows the work-precision diagram for the `accelerInt` solvers: the vertical axis shows the error (as measured by Eqs. (4.24) and (4.25)) for the solvers over the various tolerances tested, while the horizontal axis shows the mean CPU-runtime (averaged over five individual runs) on a single core of a Intel® Xeon® X5650 CPU (with SSE4.2 vector instructions), using `v16.1.1` of the Intel OpenCL runtime [132]. The `RKF45` solver was omitted from this test, as the stiffness of the chemical kinetic ODEs was such that the computational cost was prohibitively expensive; this

solver will be validated in future works for less stiff-problems.

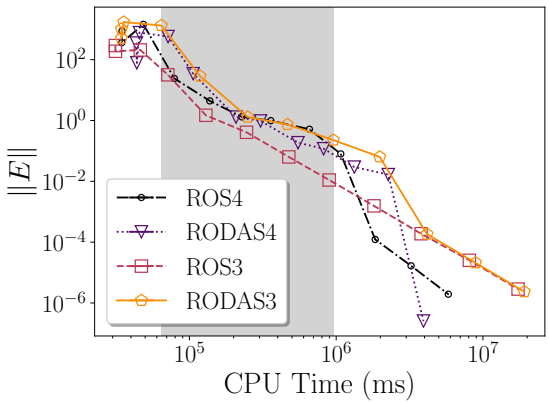



Figure 4.1: The work-precision diagram for the implicit-OpenCL solvers in `accelerInt`. The vertical axis shows the error norm computed by Eqs. (4.24) and (4.25), while the horizontal axis shows the mean runtime of the solver, measured in milliseconds. The region marked in grey corresponds to "intermediate" tolerances, ranging from $TOL$ =$10^{-7}$–$10^{-11}$ (with equal absolute and relative tolerances in the adaptive time-stepping algorithm).

For loose tolerances (i.e., $10^{-4}$–$10^{-6}$), the performance and error of all the tested solvers is very similar. However, for intermediate tolerances ($10^{-7}$–$10^{-11}$, marked in grey on Fig. 4.1) the `ROS3` solver consistently has the lowest error compared to the reference solution; the fourth order-solvers (`ROS4`, `RODAS4`) tend to be slightly faster for a minor increase in error in this region. The higher accuracy of the `ROS3` solver in this region is likely due to the use of a different set of method coefficients (see Table 4.1). At very strict tolerances, i.e., less than $10^{-11}$, the `ROS4` solver is both faster and more accurate than the third-order methods, while the `RODAS4` solver is the most accurate for the strictest tolerance (i.e., $10^{-15}$).

### 4.3.2 Constant-pressure ignition in `OpenFOAM`

To validate the coupling of the `accelerInt` solvers to the `OpenFOAM` chemistry model (see Section 4.2.4), a series of constant-pressure homogeneous ignition problems were run over a range of initial temperatures, pressures and equivalence ratios, listed in Table 4.2, using the GRI-Mech 3.0 chemical kinetic model [89] and $CH_4$ as the fuel. These conditions cover both low, intermediate and high temperature ignition, as well as lean, stoichiometic and rich fuel mixtures to test the accuracy of the solvers over different chemical regimes. To test the relative accuracy of the various solvers, the ignition problems were simulated to near chemical equilibrium (post-ignition) with both `accelerInt` and built-in `OpenFOAM` integrators. The value of the

thermochemical kinetic state-vectors for each solver was sampled at 10 individual times, equally distributed over the entire simulated time-span and excluding the initial state; an example of the time-sampling can be seen in Fig. 4.2a. Finally, the sampled values were compared to a reference solution computed by the commonly used combustion chemistry code `Cantera` [173], in order to assess their accuracy. In this example we used the fourth-order linearly-implicit Rosenbrock solver (`ROS4`) in both `OpenFOAM` and `accelerInt`, and the absolute and relative ODE integration tolerances were set to $10^{-10}$ and $10^{-6}$, respectively, for the `OpenFOAM` and `accelerInt` solvers, and to $10^{-20}$ and $10^{-15}$ for `Cantera`. Figure 4.2 compares the values of the temperature and species mass-fractions of $CH_4$, OH, and NO for several different initial conditions, showing that all three solvers are in qualitative agreement.

| Parameter | Values |
|---|---|
| $T_0$ | 850, 1100, and 1500 K |
| $P_0$ | 1, 10, and 25 atm |
| $\phi$ | 0.5, 1.0, and 1.5 |

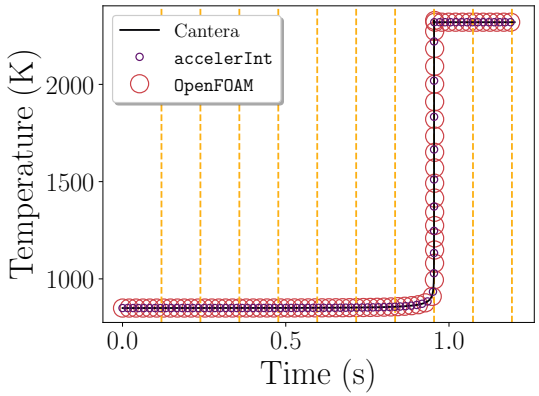
Table 4.2: A listing of initial temperatures, pressures, and $CH_4$/air equivalence ratios used for the constant-pressure homogeneous ignition problems in the validation of the `accelerInt` coupling to `OpenFOAM`. We note that the ignition delay of several of the $T = 850$ K cases was longer than 10 s; these cases: $P_0 = 1$ atm, for $\phi = 0.5$, 1.0, and 1.5, and $\phi = 0.5$, for $P_0 = 10$ and 25 atm, were not considered.

To quantify this comparison, the infimum and mean $L^2$ norms of the (filtered) relative error between the tested solver and `Cantera` were calculated as:

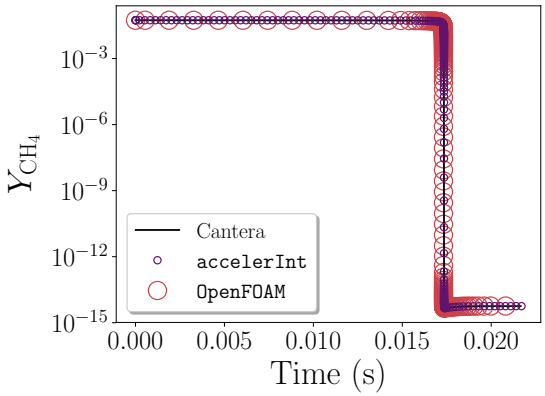
$$\|E\|_\infty = \left\| \frac{|\Phi_i^\circ(t_j) - \Phi_i(t_j)|}{|1 \times 10^{-10} + \Phi_i^\circ(t_j)|} \right\|_\infty \tag{4.26}$$

$$\|E\|_{\text{mean}} = \left\| \frac{1}{N_s(N_{\text{sp}}+1)} \frac{|\Phi_i^\circ(t_j) - \Phi_i(t_j)|}{|1 \times 10^{-10} + \Phi_i^\circ(t_j)|} \right\|_2 \tag{4.27}$$

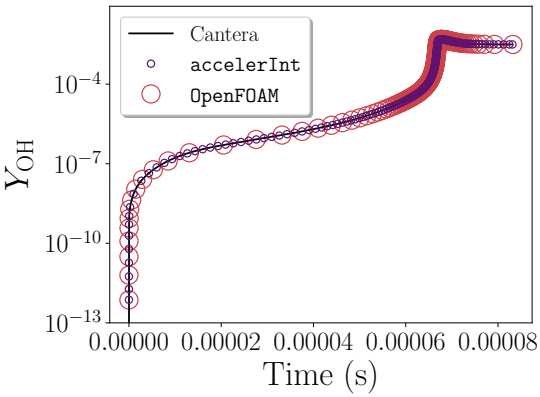
where $N_s$ is the total number of sampled points, $N_{\text{sp}} + 1$ is the size of the thermochemical state-vector used in `OpenFOAM` (i.e., the temperature and species concentrations), and $\Phi_i(t_j)$ is the $i$th entry of the state-vector calculated by either `OpenFOAM` or `accelerInt` at the $j$th sampled point, and $\Phi_i^\circ(t_j)$ the same value as calculated by `Cantera`. The values of these norms are in shown in Table 4.3 for both solvers, showing that `accelerInt` has significantly better agreement with the reference solution computed by `Cantera`. We note that although the computed error norms for `OpenFOAM` are several orders of magnitudes larger (i.e., $\mathcal{O}(10^5)$–$\mathcal{O}(10^6)$ versus $\mathcal{O}(10^{-1})$–$\mathcal{O}(1)$ for `accelerInt`), it does not imply that the `OpenFOAM` solvers is grossly



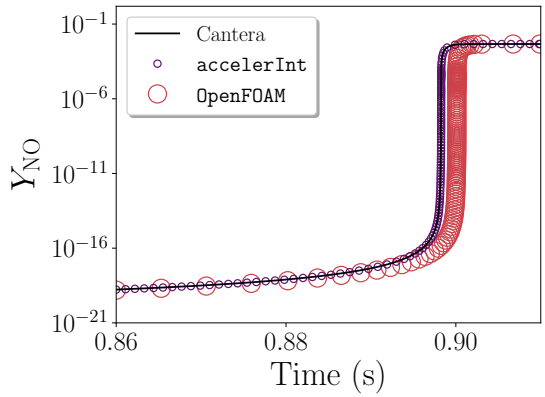

(a) The predicted temperature traces at $P_0 = 25\,\text{atm}$, $T_0 = 850\,\text{K}$ and $CH_4$/air of $\phi = 1.5$.The vertical dashed-lines show the points at which the solution was sampled for error evaluation.




(b) The $CH_4$ mass-fraction profiles predicted by the various solvers at $P_0 = 10\,\text{atm}$, $T_0 = 1100\,\text{K}$ and $CH_4$/air of $\phi = 1.0$




(c) The OH mass-fraction profiles predicted by the various solvers at $P_0 = 25\,\text{atm}$, $T_0 = 1500\,\text{K}$ and $CH_4$/air of $\phi = 0.5$




(d) A zoomed-in look at the NO mass-fraction profiles predicted by the various solvers during the ignition event, for $P_0 = 25\,\text{atm}$, $T_0 = 850\,\text{K}$ and $CH_4$/air of $\phi = 1.0$.

Figure 4.2: Comparison of constant-pressure homogeneous ignition problem with various solvers using GRI-Mech 3.0. We note that the plotted points for the `OpenFOAM` and `accelerInt` solvers were thinned somewhat for visibility.

inaccurate; indeed, a visual comparison of the solutions over the entire simulation (Figs. 4.2a to 4.2c) show no easily discernible discrepancies. Instead, this difference is largely a function of more accurate predictions of the ignition delay time on the part of `accelerInt`. For instance, Fig. 4.2d shows the predicted mass fraction of NO during ignition for the stoichiometic case with $P_0 = 25\,\text{atm}$ and $T_0 = 850\,\text{K}$; the `OpenFOAM` solver predicts ignition ~1.8 ms later than either `accelerInt` or `Cantera`, a mere 0.21 % difference. Nonetheless, we conclude that `accelerInt`'s `ROS4` solver is more accurate than the corresponding `OpenFOAM` implementation.

| Solver | $\|E\|_{\text{mean}}$ | $\|E\|_{\infty}$ |
|---|---|---|
| `OpenFOAM` | $4.68 \times 10^{5}$ | $9.49 \times 10^{6}$ |
| `accelerInt` | $3.39 \times 10^{-1}$ | $8.12 \times 10^{0}$ |

Table 4.3: The filtered mean and infimum relative error norms comparing the computed solutions of the `OpenFOAM` and `accelerInt` solvers to `Cantera` for the homogeneous constant-pressure ignition problems.

## 4.4 Sandia Flame D

### 4.4.1 Case description

Next, the precision and performance of the `OpenFOAM` and `accelerInt` solvers will be compared for a more realistic reactive-flow test case. The Sandia Flame D [152–154] is a well-characterized piloted $CH_4$/air jet flame and a turbulent Reynolds number of $Re_t = 22000$. As of `OpenFOAM v5.x` (the 2017 release from the OpenFOAM foundation [174]), a case modeling this flame is included as a tutorial, providing a relatively simple/inexpensive but more realistic proving ground for the coupled `accelerInt` solvers. The operating conditions and key dimensions of the case are listed in Table 4.4, while a schematic of the configuration super-imposed over the simulation mesh is shown in Fig. 4.3c.

| Dimension | Value |
|---|---|
| $D_{\text{jet, inner}}$ | 7.2 mm |
| $D_{\text{pilot, inner}}$ | 7.7 mm |
| $D_{\text{pilot, outer}}$ | 18.2 mm |
| $D_{\text{wall, outer}}$ | 18.9 mm |
| Exit | 30 mm×30 mm |

(a) Diameter of the jet, pilot and wall and exit-plane dimensions for the Sandia Flame D case.

| | Source | | |
|---|---|---|---|
| | Jet | Coflow | Pilot |
| Composition | $CH_4$:25 %/Air:75 %[†] | Dry air | Equil.[‡] |
| Velocity | 49.6 m/s | 0.9 m/s | 11.4 m/s |
| Temperature | 294 K | 291 K | 1880 K |
| Pressure | 0.993 atm | 0.993 atm | 0.993 atm |

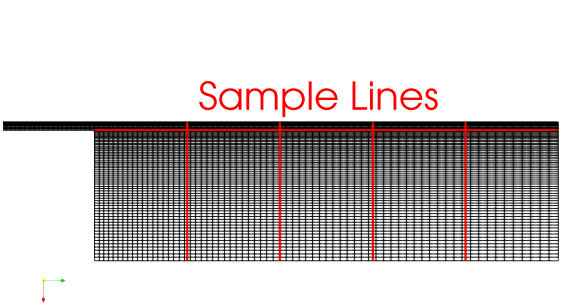
(b) Operating conditions for the Sandia Flame D case.
[†]The jet composition is measured by percent volume.
[‡]Equilibrium state of $CH_4$/air at $\phi = 0.7$

Table 4.4: Operating conditions and dimensions of the Sandia Flame D case

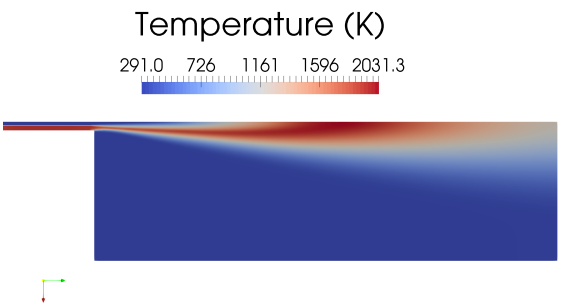
The mesh for this case, pictured in Fig. 4.3a, is fully orthogonal with 5170 cells, ranging in size

from roughly ~5 mm on a side to ~0.72 mm tall near the wall. The solution domain is a thin three-dimensional wedge, with axi-symmetric boundary conditions on the front and back faces, and zero-gradient/total-pressure boundary conditions on the outlet. The case, as packaged with `OpenFOAM`, utilizes second-order interpolation and gradient schemes for all variables, but a strongly limited scheme (tending towards first-order) for divergence calculations. In addition, a standard k-$\varepsilon$ RANS model is used to model turbulence and a pseudo-transient, first-order time-stepping scheme is used initially to advance to a steady-state solution.

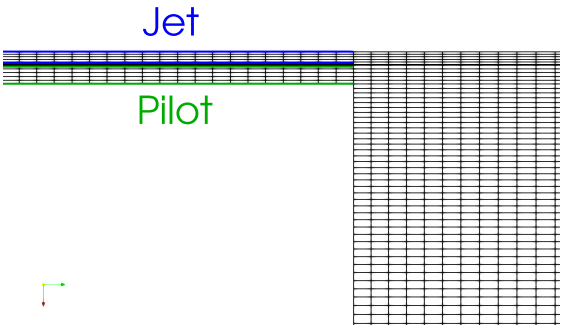
The baseline case was modified slightly to better suit the purposes of this study. First, the chemical kinetic model, originally a 36-species skeletal methane mechanism used in `OpenFOAM` was replaced with the full GRI-Mech 3.0 model. Second, the base `OpenFOAM` case utilized a tabulated dynamic adaptive chemistry scheme [38, 39] to accelerate the solution process, this has been disabled such that the performance and accuracy of the ODE solvers can be directly compared. Finally, after reaching steady-state, the time-stepping scheme was switched to second-order implicit method, the minimum reacting temperature was set to 500 K, and the case was run for an additional 10 ms of simulated time using a CFD time-step of $\Delta t = 10^{-6}$ sec.




(a) The mesh for the Sandia Flame D case. The red line denotes where the solution was sampled for validation.




(b) The steady-state temperature profile in K, as solved for by the base `OpenFOAM` fourth-order Rosenbrock ODE integrator.



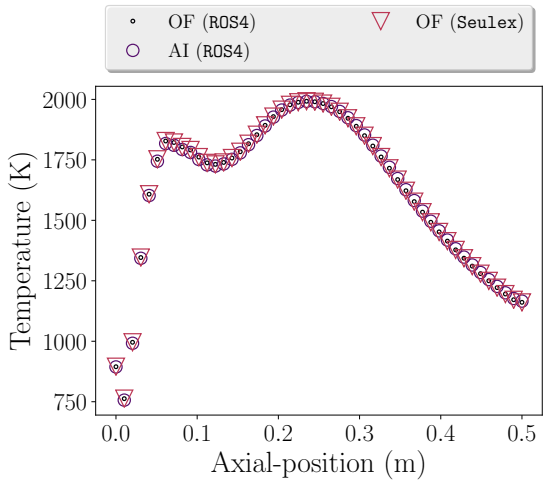

(c) A close-up view of the inlet to the domain. The jet and pilot flames are marked in blue and green, respectively. The sample lines are marked in red, as in Fig. 4.3a

Figure 4.3: The mesh and steady-state temperature-profile of the Sandia Flame D case.

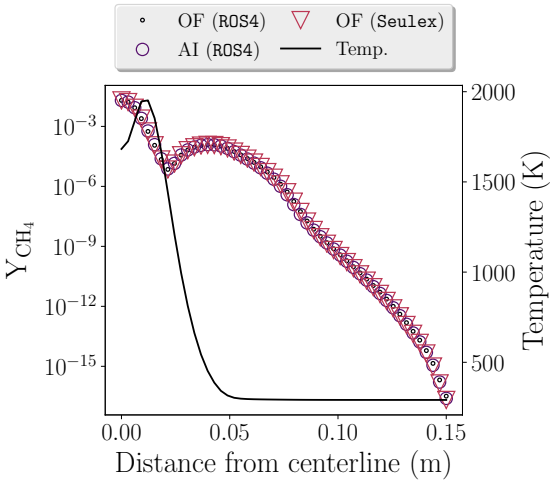
### 4.4.2 Verificaton

To compare the predicted solutions of the `OpenFOAM` and `accelerInt` solvers in this setting, the solution state was sampled along several lines (pictured in Fig. 4.3a) cutting through the plane of the flame (Fig. 4.3b) both vertically and axially. The vertical sample lines are evenly spaced at intervals of 0.1 m extending downwards from the center-line to the bottom of the domain, with the first occurring at 0.1 m from the end of the jet; the axial sample line extends from the nozzle to the far right end of the domain. In Figs. 4.4a to 4.4d the steady-state and time-dependent (sampled at the final state of $t = 0.01$ s) temperature and mass-fraction profiles of $CH_4$, OH and NO across the flame are compared for three different solvers: `accelerInt`'s 4th-order linearly-implicit Rosenbrock method (`ROS4`), `OpenFOAM`'s `ROS4` solver, and `OpenFOAM`'s `Seulex` implementation, a linearly-implicit extrapolation Euler integration method [67]. Overall there is excellent qualitative and quantitative agreement between all three solvers for each quantity, demonstrating that `accelerInt`-coupling can accurately reproduce the solutions of the built-in `OpenFOAM` solver for temperature, as well as major and minor species. However, certain discrepancies do exist between the predicted solutions. For example, in Fig. 4.4c it is seen that the steady-state solution computed by `OpenFOAM`'s `Seulex` solver predicts larger amounts of OH near the wall of the domain (i.e., at more than ~0.1 m from the center-line), while the `OpenFOAM` `ROS4` solver is in closer agreement with `accelerInt`. We note that for this sample, the temperature is lower than 500 K for all points more than ~0.06 m from the center-line, hence the solution in this region is dominated by convective and mixing processes, as chemical reactions are largely inactive at such low temperatures. This demonstrates that even for two ODE integrators—i.e., `OpenFOAM`'s `ROS4` and `Seulex` methods—using identical tolerances and implementation of the chemical kinetic source-terms and Jacobian matrix, it is very difficult to predict how small discrepancies in the integrated thermochemical state vectors (i.e., those computed in higher-temperature regions where chemistry is more active) might grow or interact over thousands of CFD time-steps, each of which use and modify the state-vectors in complex (often non-linear) numerical methods.

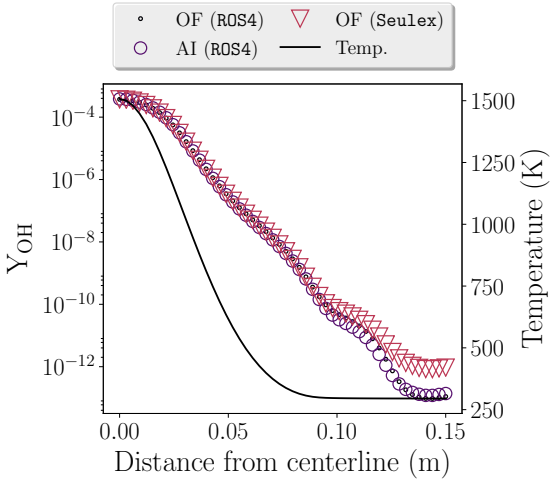
Nonetheless, we will use the norms from Eqs. (4.26) and (4.27) to obtain a sense of the similarity of the solutions computed by the various solvers. However, here these norms will be labeled the mean and maximum percent-differences, both because we do not have a true reference solution to compare to, but also to re-emphasize that minor-differences in the solutions are to be expected when coupled to a full CFD solver. Additionally, as the `accelerInt` solver was shown to be in far better agreement with `Cantera` for the constant-pressure homogeneous ignition problems (Section 4.3.2), we will use `accelerInt` as the "reference" solution for Eqs. (4.26) and (4.27). Further, when computing these percent differences, all points where the temperature (as



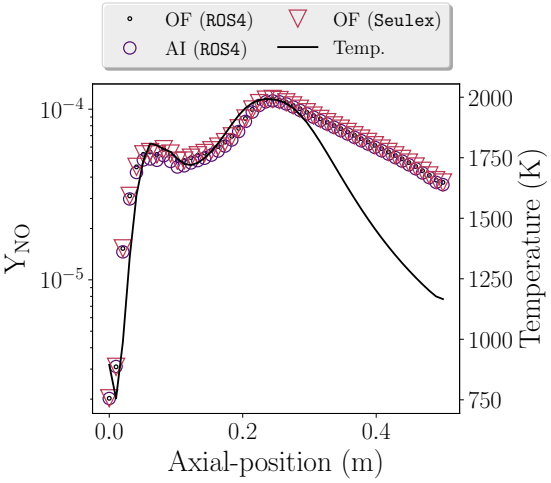

(a) The steady-state temperature solution profiles along the axial sample line extending from the nozzle to the edge of the domain.




(b) The time-dependent solution profiles of the mass fraction of $CH_4$ sampled at 0.2 m away from the nozzle (i.e., the second sample line in Fig. 4.3c) for $t = 0.01$ s. The temperature of the flame along the same line is plotted as well to give a sense of the flame-width, obtained using `AI (ROS4)`.




(c) The steady-state solution profiles of the mass fraction of OH sampled at 0.4 m away from the nozzle (i.e., the fourth sample line in Fig. 4.3c). The temperature of the flame along the same line is plotted as well to give a sense of the flame-width, obtained using `AI (ROS4)`.




(d) The time-dependent solution profiles of the mass fraction of NO along the axial sample line extending from the nozzle to the edge of the domain for $t = 0.01$ s. The temperature of the flame along the same line is plotted as well for comparison purposes, obtained using `AI (ROS4)`..

Figure 4.4: Temperature and species profile comparisons for both the steady-state and time-dependent solutions, along various sampling lines pictured in Fig. 4.3a. "OF" denotes an `OpenFOAM` solver, while "AI" marks the `accelerInt` version.

predicted by `accelerInt`) was less than 500 K were excluded as the chemistry is unimportant in these locations, and differences were observed (Fig. 4.4c) even among the `OpenFOAM` solvers, as previously discussed. In Table 4.5, we see that the mean percent-difference between the `OpenFOAM` solvers and the `accelerInt` solver is roughly ~5 %, while the maximum percent difference reaches ~34 % in some cases. However, the maximum percent-difference in temperature and pressure is significantly lower, at just ~0.8 % and $6 \times 10^{-4}$ %, respectively. It is noted that if we compare the `OpenFOAM` solvers directly—that is, if the `OpenFOAM ROS4` solver is used for the reference solution in calculation of the percent-difference norms—the maximum and mean percent-difference of the `OpenFOAM Seulex` solver are roughly ~6 % and ~0.3 %, respectively. Once again, this demonstrates that even while solving chemical kinetic ODEs with the same implementation of the chemistry evaluations and nominal tolerances, changing the integration method may lead to slightly different answers. Thus we conclude that the overall agreement between all three solvers is deemed acceptable.

| | Steady-state | | Time-dependent | |
|---|---|---|---|---|
| Solver | `OpenFOAM-Seulex` | `OpenFOAM-ROS4` | `OpenFOAM-Seulex` | `OpenFOAM-ROS4` |
| $\|D\|_{\text{mean}}$ | $5.34 \times 10^{0}$ % | $5.38 \times 10^{0}$ % | $5.60 \times 10^{0}$ % | $5.63 \times 10^{0}$ % |
| $\|D\|_{\infty}$ | $2.78 \times 10^{1}$ % | $2.79 \times 10^{1}$ % | $3.43 \times 10^{1}$ % | $3.40 \times 10^{1}$ % |
| $\|D\|_{T,\infty}$ | $8.15 \times 10^{-1}$ % | $8.04 \times 10^{-1}$ % | $8.22 \times 10^{-1}$ % | $8.09 \times 10^{-1}$ % |
| $\|D\|_{p,\infty}$ | $3.86 \times 10^{-4}$ % | $3.97 \times 10^{-4}$ % | $5.63 \times 10^{-4}$ % | $6.22 \times 10^{-4}$ % |

Table 4.5: Comparison of the mean and maximum percent-differences of the `OpenFOAM` solvers to `accelerInt`, using the norms defined analogously to those in Eqs. (4.26) and (4.27). The $\|D\|_{T,\infty}$ and $\|D\|_{p,\infty}$ values are the maximum percent differences in the temperature and pressure, respectively.

### 4.4.3 Performance

Next, the performance of the various `OpenFOAM` and `accelerInt` solvers will be tested using the Sandia Flame D case. The performance studies were run on 10 cores of an Intel E5-2690 V3 CPU, with `AVX2` vector instructions, 128 Gbit of RAM and `v16.1.1` of the Intel OpenCL runtime. `OpenFOAM` version `6.x` was used with `v2.1.0` of the OpenMPI library [175], and was compiled with `gcc v5.4.0` [176]. The `reactingFoam` solver was instrumented with the MPI-profiling library `IPM v2.0.6` [177], by placing profiling sections (via `MPI_PControl` calls) around the calls to the turbulent combustion model, ODE integration, and other key parts of the CFD-timestep (e.g., convection evaluation).

Figure 4.5 shows the total wall-clock execution time (over all 10 processors) spent by each solver integrating the chemical kinetic ODEs. The `accelerInt` solver spends just 10.2–31.1 h solving chemistry for the steady-state and time-dependent cases, respectively, while both the `OpenFOAM` solvers take over 100 h in all cases. The slowest integration method is the `OpenFOAM Seulex`

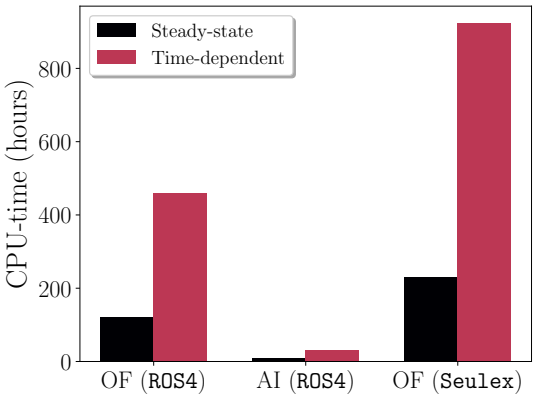



Figure 4.5: Total CPU-hours spent evaluating chemistry in the Sandia Flame D case.

| | Solver | | |
|---|---|---|---|
| | AI (ROS4) | OF (ROS4) | OF (Seulex) |
| Steady-state | 12.0 × | – | 0.53 × |
| Time-dependent | 14.8 × | – | 0.5 × |
| Chemistry time | 93.9–95.8 % | 99.3–99.6 % | 99.7–99.9 % |

Table 4.6: Speeds of the chemistry solvers, normalized by the OpenFOAM ROS4 solver and the percent of total execution time spent integrating the chemical-kinetic ODEs.

solver, which takes over 923 h to complete the time-dependent solution. We note that here, chemistry evaluation time includes time-spent idle due to the poor chemistry-load balancing present in most OpenFOAM simulations*.

The speeds of the solvers are compared using the OpenFOAM ROS4 solver as the baseline in Table 4.6. The accelerInt ROS4 solver ranges from 12–14.8 × faster than the OpenFOAM equivalent. In addition, the OpenFOAM Seulex solver is roughly 2 × slower than the OpenFOAM ROS4 solver in both cases. Finally, the accelerInt solver spends ~94–96 % in chemistry integration, while both OpenFOAM methods use over 99 % of the run-time solving chemistry.

## 4.5 Conclusions

In this effort, several previously developed [57] linearly-implicit and explicit OpenCL-vectorized ODE-integration methods were incorporated into the accelerInt library and used to accelerate chemical kinetic integration in a reactive-flow simulation. The major contributions of this work were:

- Adaption of the accelerInt solvers to use the chemical kinetic source-rate and analytical Jacobian evaluation codes from the pyJac code-generation platform [149];

*OpenFOAM uses a simple static decomposition of the domain, which is to say there is no chemistry load-balancing occurring.

- Validation of the solvers against the commonly used implicit integrator `CVODE` [42, 150];
- Extension of the `OpenFOAM` CFD-code to utilize vectorized-ODE integration techniques and implementation of a species-mole based EDC turbulent combustion model to accommodate `pyJac`'s thermochemical state-vector; and finally
- Validation and performance comparisons of the `accelerInt` `ROS4` solver to two built-in `OpenFOAM` integration techniques, `ROS4` and `Seulex`.

This work demonstrates that vectorized chemical kinetic integration paired with analytical Jacobian and source-rate evaluation can realize speedups over an order of magnitude (12–15 $\times$) on the same CPU for realistic reactive-flow simulations using detailed chemical kinetic models in `OpenFOAM`. Further, the `accelerInt` solvers were shown to accurately solve the chemical kinetic ODEs for a variety of cases in this effort.

Future extensions to this work should focus on a few key aspects. First, a study of the `accelerInt` solvers in a larger scale reactive flow simulation than the current Sandia Flame D case—modeling of the Volvo bluff-body premixed turbulent reacting flame [178, 179]—is currently being conducted. Since the use of a steady-state solution to initialize the time-dependent simulations in the current Sandia Flame D case may lead to predicted temperature and species profiles that are not particularly sensitive to the accuracy of the ODE-solver used, other cases more sensitive to the effect of ODE-solver accuracy on the predicted solution should be investigated. In addition, this extension aims to determine the effect of use of a high-resolution mesh and an LES turbulence model on the performance of the `accelerInt` and `OpenFOAM` solvers, as well chemistry load-balancing, parallel scaling efficiencies, and other performance concerns.

More generally, this work has shown that vectorized linearly-implicit ODE-integration (paired with analytical Jacobian evaluation) can greatly accelerate reactive-flow simulations. The development of vectorized sparse linear-algebra/matrix factorization codes would be an excellent—if challenging—addition to this result, further speeding up the integration of larger detailed chemical models. Finally, more advanced integration algorithms should be investigated; W-methods [67] are particularly promising for their ability to re-use previously evaluated Jacobians and LU-factorizations without need for Newton-iteration.

# Chapter 5

# Conclusions and recommendations for future work

## 5.1 Summary

The research in this dissertation focused on the goal of reducing the cost of using accurate chemical kinetics in realistic reactive-flow simulations.

In Chapter 2, several GPU-based methods for the integration of stiff chemical kinetic ODEs were validated, and their performance compared to the commonly used implicit CPU solver `CVODEs` for two chemical kinetic models [88, 89] and different levels of numerical stiffness. The GPU-based implicit Runge–Kutta method, `Radau-IIa`, was equivalent to `CVODEs` running on 12–38 CPU cores for the less stiff, smaller global time-step size ($\Delta t = 10^{-6}$ s), but the performance degraded to just 3–15 cores for the larger global time-step size ($\Delta t = 10^{-4}$ s) due to thread-divergence and memory traffic concerns. Additionally, the exponential solvers were less efficient than both the CPU and GPU implicit integrators for the tested cases.

In Chapter 3, the analytical chemical kinetic Jacobian generation platform `pyJac` was extended to enable both SIMD and SIMT-vectorized evaluation on the CPU, GPU and other accelerators. A new thermochemical state vector was utilized, resulting in a Jacobian formulation that greatly increased the matrix sparsity; correspondingly, the ability to use a sparse matrix format was added to `pyJac`. In addition, we demonstrated significant speedups—up to 4.1–9.4 ×—over a strictly parallel evaluation of the chemical kinetic source-rates and Jacobian on the CPU. Further, the shallow-vectorized OpenCL codes were faster than a previous version of `pyJac` over all tested chemical kinetic models [88, 89, 135, 136], reaching speedups of 19.56 ×.

Finally, in Chapter 4, the methods developed in this thesis were applied to a realistic

reactive-flow simulation using the `OpenFOAM` CFD code. First, several linearly-implicit vectorized ODE integration methods were incorporated into the `accelerInt` software package, and validated against `CVODEs`. Next, `OpenFOAM` was extended to incorporate the vectorized solvers, and an EDC turbulent combustion model utilizing species moles (as used in `pyJac`) was implemented. In addition, the vectorized solvers were found to be significantly more accurate than an ODE integrator built into `OpenFOAM` for zero-dimensional homogeneous ignition cases, as compared to the commonly used chemical kinetics code `Cantera`. Lastly, the performance and precision of the vectorized solvers was compared to two `OpenFOAM` integrators for a simulation modeling the Sandia Flame D; all three solvers were found to be in good agreement with each other, while the vectorized solver was 12–15 $\times$ faster than the two `OpenFOAM` solvers for steady-state and time-dependent simulations.

## 5.2 Directions for future work

Although we have demonstrated that significant speedups can be achieved via the use of vectorized chemical kinetic integration algorithms paired with analytical chemical kinetic Jacobians in realistic reactive-flow simulations, further accelerations could be achieved by progress in a few key areas.

First, the ability to cheaply and accurately estimate the expected level of numerical stiffness that will be encountered during the integration of the chemical kinetic ODEs starting from a given thermochemical state—and further, correlating this stiffness measure to the performance of various ODE integration methods on different platforms—could significantly bring down the cost of using detailed chemical kinetic models in realistic reactive-flow simulations. It has been previously shown that GPU-based stabilized explicit integration methods are capable of achieving order-of-magnitude (or greater) speedups [49, 50, 52], as compared to standard implicit CPU-based methods, for the integration of non-stiff to moderately stiff chemical kinetics. Additionally, while Chapter 2 demonstrated an implicit GPU-based integrator method that accelerated the solution of the chemical kinetic ODEs in some cases, thread-divergence related performance degradation due to high-levels of numerical stiffness remained a challenging issue. In contrast, in Chapter 4, we showed that significant speedups can be achieved over existing CPU-based linearly-implicit integration techniques using SIMD-vectorized ODE integration and analytical Jacobian techniques. What is truly needed, in our opinion, is a way to combine the strengths of explicit and implicit/linearly-implicit solution techniques on both the CPU and GPU. Such a method would function similarly to a load-balancing algorithm for chemical kinetics, except it would additionally consider the numerical stiffness (and resulting

computational cost for various ODE integration methods), vectorization capabilities, and presence of hardware accelerators (e.g., GPUs) to maximize performance. For instance, when using a heterogeneous-platform-/vectorization- aware chemical kinetic integration load-balancing algorithm, one could imagine solving large numbers of moderately stiff chemical kinetic IVPs (e.g., near-equilibrium states inside the flame) on a small number of GPUs, while CPU-based vectorized methods (or even traditional CPU-based high-order implicit schemes) solve relatively fewer highly-stiff chemical kinetic IVPs. To our knowledge, there have been just a few studies that have attempted to implement similar techniques [50, 180], however the stiffness detection and computation cost estimation methods tended to be empirical in nature and it is not clear how well they would generalize to multiple integration algorithms, vectorization patterns, and hardware platforms. In addition, there has been some recent work [181] on the classification and estimation of numerical stiffness encountered during the integration of chemical kinetic ODEs, and the relationship to computational cost for various integration methods. However, the interactions of the encountered numerical stiffness, desired accuracy, and selected integration algorithm are quite complex, and making it difficult to predict integrator performance. Therefore, more work is needed on this front to develop a robust prediction methodology to power the load-balancing algorithm.

The second major thrust of future work should be focused on the development of vectorized sparse-linear algebra and matrix factorization techniques, these methods have been shown to greatly accelerate linear-algebra operations in a serial CPU context [182] and could enable the use of significantly larger chemical kinetic models in reactive-flow simulations. We note that many mature libraries, e.g., [148, 182], exist for these types of operations for serial execution on the CPU, however equivalent libraries for vectorized execution [147, 183] are often optimized for use of single large matrix (instead of operation over many relatively smaller matrices) or lack implementations of key algorithms, e.g., LU-factorization. Indeed, vectorized sparse matrix factorization tends to be difficult due both to the irregular structure of the matrix and dependence of the resulting factorization on the values of the matrix to be factorized [184]. Nonetheless, the extent to which existing codes may be adopted to enable sparse matrix and linear-algebra operations for vectorized chemical kinetic ODE integration should be assessed, and missing pieces developed. In addition, the speedups that can be achieved using `pyJac`'s sparse chemical kinetic Jacobian coupled with standard sparse linear algebra libraries [148, 182] and CPU-based implicit algorithms (e.g., `CVODEs`) should be investigated.

Finally, as demonstrated in Chapter 4, significant speedups can be achieved by the used of SIMD-vectorized linearly implicit integration techniques for reactive-flow simulations, thus we recommend development of vectorized versions of more advanced linearly-implicit solvers to

further increase performance. For instance, high-order W-methods [67, 185, 186] are particularly promising in that they do not require an exact Jacobian at each internal integration time-step, and thus previous Jacobian evaluations (and importantly, LU-factorizations) could be reused for multiple time-steps, reducing computational cost. In addition, multi-step Rosenbrock-type "peer" methods can help avoid order-reduction for very stiff ODEs [186, 187], and would be easily vectorized as each stage of the integrator is computed independently. Use of these more complex methods could further increase CPU-based vectorized ODE integrator performance.

# Bibliography


[1] United States Energy Information Administration. *Monthly Energy Review*. July 2017.

[2] United States Energy Information Administration. *Annual Energy Outlook*. Feb. 2018.

[3] S. Imtenan et al. "Impact of low temperature combustion attaining strategies on diesel engine emissions for diesel and biodiesels: A review". In: *Energy Convers. Manag.* 80 (2014), pp. 329–356. ISSN: 0196-8904. DOI: 10.1016/j.enconman.2014.01.020.

[4] A. Cavaliere and M. de Joannon. "Mild Combustion". In: *Progress Energy Combust. Sci.* 30.4 (2004), pp. 329–366. ISSN: 0360-1285. DOI: 10.1016/j.pecs.2004.02.003.

[5] C. K. Westbrook et al. "Computational combustion". In: *Proc. Combust. Inst.* 30.1 (2005), pp. 125–157. ISSN: 1540-7489. DOI: 10.1016/j.proci.2004.08.275.

[6] N. Komninos and C. Rakopoulos. "Modeling HCCI combustion of biofuels: A review". In: *Renew. Sustain. Energy Rev.* 16.3 (2012), pp. 1588–1610. ISSN: 1364-0321. DOI: 10.1016/j.rser.2011.11.026.

[7] T. Lu and C. K. Law. "Toward accommodating realistic fuel chemistry in large-scale computations". In: *Prog. Energy Combust. Sci.* 35.2 (2009), pp. 192–215. ISSN: 0360-1285. DOI: 10.1016/j.pecs.2008.10.002.

[8] S. M. Sarathy et al. "Comprehensive chemical kinetic modeling of the oxidation of 2-methylalkanes from $C_7$ to $C_{20}$". In: *Combust. Flame* 158.12 (Dec. 2011), pp. 2338–2357. DOI: 10.1016/j.combustflame.2011.05.007.

[9] M. Mehl et al. "Kinetic modeling of gasoline surrogate components and mixtures under engine conditions". In: *Proc. Combust. Inst.* 33.1 (2011), pp. 193–200. DOI: 10.1016/j.proci.2010.05.027.

[10] M. Mehl et al. "An Approach for Formulating Surrogates for Gasoline with Application toward a Reduced Surrogate Mechanism for CFD Engine Modeling". In: *Energy Fuels* 25.11 (2011), pp. 5215–5223. DOI: 10.1021/ef201099y.

[11] O. Herbinet, W. J. Pitz, and C. K. Westbrook. “Detailed chemical kinetic mechanism for the oxidation of biodiesel fuels blend surrogate”. In: *Combust. Flame* 157.5 (2010), pp. 893–908. DOI: 10.1016/j.combustflame.2009.10.013.

[12] C. Huang et al. “Study of dimethyl ether homogeneous charge compression ignition combustion process using a multi-dimensional computational fluid dynamics model”. In: *Int. J. Therm. Sci.* 48.9 (2009), pp. 1814–1822. ISSN: 1290-0729. DOI: 10.1016/j.ijthermalsci.2009.02.006.

[13] F. Bottone et al. “The Numerical Simulation of Diesel Spray Combustion with LES-CMC”. In: *Flow, Turbul. Combust.* 89.4 (2012), pp. 651–673. ISSN: 1573-1987. DOI: 10.1007/s10494-012-9415-y.

[14] A. A. Moiz et al. “Study of soot production for double injections of n-dodecane in CI engine-like conditions”. In: *Combust. Flame* 173 (2016), pp. 123–131. ISSN: 0010-2180. DOI: 10.1016/j.combustflame.2016.08.005.

[15] A. C. Hindmarsh and R. Serban. *CVODE v2.8.2*. http://computation.llnl.gov/projects/sundials-suite-nonlinear-differential-algebraic-equation-solvers/download/cvode-2.8.2.tar.gz. Aug. 2015.

[16] T. Lu and C. K. Law. “Toward accommodating realistic fuel chemistry in large-scale computations”. In: *Prog. Energy Comb. Sci.* 35.2 (2009), pp. 192–215. DOI: 10.1016/j.pecs.2008.10.002.

[17] T. Turányi and A. S. Tomlin. *Analysis of Kinetic Reaction Mechanisms*. Berlin Heidelberg: Springer-Verlag, 2014. ISBN: 978-3-662-44561-7. DOI: 10.1007/978-3-662-44562-4.

[18] T. Lu and C. K. Law. “Linear time reduction of large kinetic mechanisms with directed relation graph: *n*-heptane and iso-octane”. In: *Combust. Flame* 144.1-2 (2006), pp. 24–36.

[19] P. Pepiot-Desjardins and H. Pitsch. “An efficient error-propagation-based reduction method for large chemical kinetic mechanisms”. In: *Combust. Flame* 154.1-2 (2008), pp. 67–81. ISSN: 0010-2180. DOI: 10.1016/j.combustflame.2007.10.020.

[20] V. Hiremath, Z. Ren, and S. B. Pope. “A greedy algorithm for species selection in dimension reduction of combustion chemistry”. In: *Combust. Theor. Model.* 14.5 (2010), pp. 619–652. DOI: 10.1080/13647830.2010.499964.

[21] K. E. Niemeyer, C. J. Sung, and M. P. Raju. “Skeletal mechanism generation for surrogate fuels using directed relation graph with error propagation and sensitivity analysis”. In: *Combust. Flame* 157.9 (2010), pp. 1760–1770. DOI: 10.1016/j.combustflame.2009.12.022.

[22] T. Lu and C. K. Law. “Diffusion coefficient reduction through species bundling”. In: *Combust. Flame* 148.3 (2007), pp. 117–126. DOI: 10.1016/j.combustflame.2006.10.004.

[23] S. S. Ahmed et al. “A comprehensive and compact *n*-heptane oxidation model derived using chemical lumping”. In: *Phys. Chem. Chem. Phys.* 9.9 (2007), pp. 1107–1126. DOI: 10.1039/b614712g.

[24] P. Pepiot-Desjardins and H. Pitsch. “An automatic chemical lumping method for the reduction of large chemical kinetic mechanisms”. In: *Combust. Theor. Model.* 12.6 (2008), pp. 1089–1108. DOI: 10.1080/13647830802245177.

[25] U. Maas and S. B. Pope. “Simplifying chemical kinetics: intrinsic low-dimensional manifolds in composition space”. In: *Combust. Flame* 88.3-4 (1992), pp. 239–264.

[26] S.-H. Lam and D. A. Goussis. “The CSP Method for Simplying Kinetics”. In: *Int. J. Chem. Kinet.* 26.4 (1994), pp. 461–486. DOI: 10.1002/kin.550260408.

[27] T. Lu, Y. Ju, and C. K. Law. “Complex CSP for chemistry reduction and analysis”. In: *Combust. Flame* 126.1-2 (2001), pp. 1445–1455. ISSN: 0010-2180. DOI: 10.1016/S0010-2180(01)00252-8.

[28] X. Gou et al. “A dynamic multi-timescale method for combustion modeling with detailed and reduced chemical kinetic mechanisms”. In: *Combust. Flame* 157.6 (2010), pp. 1111–1121. DOI: 10.1016/j.combustflame.2010.02.020.

[29] T. Lu and C. K. Law. “Strategies for mechanism reduction for large hydrocarbons: n-heptane”. In: *Combust. Flame* 154 (2008), pp. 153–163. ISSN: 0010-2180. DOI: 10.1016/j.combustflame.2007.11.013.

[30] K. E. Niemeyer and C. J. Sung. “Mechanism reduction for multicomponent surrogates: A case study using toluene reference fuels”. In: *Combust. Flame* 161.11 (2014), pp. 2752–2764. DOI: 10.1016/j.combustflame.2014.05.001.

[31] K. E. Niemeyer and C. J. Sung. “Reduced chemistry for a gasoline surrogate valid at engine-relevant conditions”. In: *Energy Fuels* 29.2 (2015), pp. 1172–1185. DOI: 10.1021/ef5022126.

[32] L. Liang, J. Stevens, and J. T. Farrell. “A dynamic adaptive chemistry scheme for reactive flow computations”. In: *Proc. Combust. Inst.* 32.1 (2009), pp. 527–534. DOI: 10.1016/j.proci.2008.05.073.

[33] Y. Shi et al. “Acceleration of the chemistry solver for modeling DI engine combustion using dynamic adaptive chemistry (DAC) schemes”. In: *Combust. Theor. Model.* 14.1 (2010), pp. 69–89. DOI: 10.1080/13647830903548834.

[34] X. Gou et al. “A dynamic adaptive chemistry scheme with error control for combustion modeling with a large detailed mechanism”. In: *Combust. Flame* 160.2 (2013), pp. 225–231. ISSN: 0010-2180. DOI: 10.1016/j.combustflame.2012.10.015.

[35] H. Yang et al. “Dynamic adaptive chemistry for turbulent flame simulations”. In: *Combust. Theor. Model.* 1 (), pp. 167–183. DOI: 10.1080/13647830.2012.733825.

[36] N. J. Curtis, K. E. Niemeyer, and C.-J. Sung. “An automated target species selection method for dynamic adaptive chemistry simulations”. In: *Combust. Flame* 162.4 (2015), pp. 1358–1374. DOI: 10.1016/j.combustflame.2014.11.004.

[37] S. B. Pope. “Computationally efficient implementation of combustion chemistry using in situ adaptive tabulation”. In: *Combust. Theor. Model.* 1.1 (1997), pp. 41–63. DOI: 10.1080/713665229.

[38] Z. Ren et al. “The use of dynamic adaptive chemistry and tabulation in reactive flow simulations”. In: *Combust. Flame* 161.1 (2014), pp. 127–137. DOI: 10.1016/j.combustflame.2013.08.018.

[39] Z. Li et al. “Assessment of On-the-Fly Chemistry Reduction and Tabulation Approaches for the Simulation of Moderate or Intense Low-Oxygen Dilution Combustion”. In: *Energy Fuels* 32.10 (2018), pp. 10121–10131. DOI: 10.1021/acs.energyfuels.8b01001.

[40] C. F. Curtiss and J. O. Hirschfelder. “Integration of stiff equations”. In: *Proceedings of the National Academy of Sciences of the United States of America* 38.3 (1952), pp. 235–243.

[41] G. D. Byrne and A. C. Hindmarsh. “Stiff ODE solvers: a review of current and coming attractions”. In: *J. Comput. Phys.* 70.1 (1987), pp. 1–62. ISSN: 0021-9991. DOI: 10.1016/0021-9991(87)90001-5.

[42] P. N. Brown, G. D. Byrne, and A. C. Hindmarsh. “VODE: a Variable-Coefficient ODE Solver”. In: *SIAM J. Sci. Stat. Comput.* 10.5 (1989), pp. 1038–1051. DOI: 10.1137/0910062.

[43] A. C. Hindmarsh et al. “SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers”. In: *ACM T. Math. Softw.* 31.3 (Sept. 2005), pp. 363–396. DOI: 10.1145/1089014.1089020.

[44] K. E. Niemeyer, N. J. Curtis, and C.-J. Sung. “`pyJac`: analytical Jacobian generator for chemical kinetics”. In: *Comput. Phys. Comm.* (Feb. 2017). ISSN: 0010-4655. DOI: 10.1016/j.cpc.2017.02.004.

[45] H. N. Khan, D. A. Hounshell, and E. R. Fuchs. “Science and research policy at the end of Moore’s law”. In: *Nat. Electron.* 1.1 (2018), pp. 14–21. DOI: 10.1038/s41928-017-0005-9.

[46] E. S. Oran and J. P. Boris. *Numerical Simulation of Reactive Flow*. 2nd. Cambridge: Cambridge University Press, 2001.

[47] K. Spafford et al. "Accelerating S3D: A GPGPU Case Study". In: *Euro-Par 2009 Parallel Process. Workshops, LNCS 6043*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 122–131. DOI: 10.1007/978-3-642-14122-5_16.

[48] Y. Shi et al. "Redesigning combustion modeling algorithms for the Graphics Processing Unit (GPU): Chemical kinetic rate evaluation and ordinary differential equation integration". In: *Combust. Flame* 158.5 (2011), pp. 836–847. DOI: 10.1016/j.combustflame.2011.01.024.

[49] K. E. Niemeyer et al. "Turbulence-chemistry closure method using graphics processing units: a preliminary test". In: *Fall 2011 Technical Meeting of the Eastern States Section of the Combust. Institute*. Storrs, CT, USA. DOI: 10.6084/m9.figshare.3384964.

[50] Y. Shi et al. "Accelerating multi-dimensional combustion simulations using GPU and hybrid explicit/implicit ODE integration". In: *Combust. Flame* 159.7 (2012), pp. 2388–2397. DOI: 10.1016/j.combustflame.2012.02.016.

[51] C. P. Stone and R. L. Davis. "Techniques for solving stiff chemical kinetics on graphical processing units". In: *J. Propul. Power* 29.4 (2013), pp. 764–773. DOI: 10.2514/1.B34874.

[52] K. E. Niemeyer and C.-J. Sung. "Accelerating moderately stiff chemical kinetics in reactive-flow simulations using GPUs". In: *J. Comput. Phys.* 256 (2014), pp. 854–871. DOI: 10.1016/j.jcp.2013.09.025.

[53] F. Sewerin and S. Rigopoulos. "A methodology for the integration of stiff chemical kinetics on GPUs". In: *Combust. Flame* 162.4 (2015), pp. 1375–1394. DOI: 10.1016/j.combustflame.2014.11.003.

[54] N. J. Curtis, K. E. Niemeyer, and C.-J. Sung. "An investigation of GPU-based stiff chemical kinetics integration methods". In: *Combustion and Flame* 179 (2017), pp. 312–324. ISSN: 0010-2180. DOI: 10.1016/j.combustflame.2017.02.005.

[55] A. Kroshko and R. J. Spiteri. "Efficient SIMD solution of multiple systems of stiff IVPs". In: *J. Comput. Sci* 4.5 (2013), pp. 377–385. DOI: 10.1016/j.jocs.2012.08.017.

[56] J. C. Linford and A. Sandu. "Chemical kinetics on multi-core SIMD architectures". In: *Proc. 9th Int. Conf. Comput. Sci.* 2009.

[57] C. P. Stone, A. T. Alferman, and K. E. Niemeyer. "Accelerating finite-rate chemical kinetics with coprocessors: comparing vectorization methods on GPUs, MICs, and CPUs". In: *Comput. Phys. Comm.* 226 (2018), pp. 18–29. DOI: 10.1016/j.cpc.2018.01.015.

[58] J. Nickolls et al. “Scalable Parallel Programming with CUDA”. In: *ACM Queue* 6.2 (Mar. 2008), pp. 40–53. DOI: 10.1145/1365490.1365500.

[59] NVIDIA. *CUDA C Programming Guide, version 7.5*. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Sept. 2015.

[60] N. J. Curtis, K. E. Niemeyer, and C. J. Sung. *Data, plotting scripts, and figures for “An investigation of GPU-based stiff chemical kinetics integration methods”*. Figshare, CC-BY license. 2017. DOI: 10.6084/m9.figshare.4596847.

[61] F. A. Cruz, S. K. Layton, and L. A. Barba. “How to obtain efficient GPU kernels: An illustration using FMM & FGT algorithms”. In: *Comput. Phys. Comm.* 182.10 (Oct. 2011), pp. 2084–2098. DOI: 10.1016/j.cpc.2011.05.002.

[62] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra. “Graphics processing unit (GPU) programming strategies and trends in GPU computing”. In: *J. Parallel Distrib. Comput.* 73.1 (2013), pp. 4–13. DOI: 10.1016/j.jpdc.2012.04.003.

[63] K. E. Niemeyer and C. J. Sung. “Recent progress and challenges in exploiting graphics processors in computational fluid dynamics”. In: *J. Supercomput.* 67.2 (Feb. 2014), pp. 528–564. DOI: 10.1007/s11227-013-1015-7.

[64] R. J. Kee, F. M. Rupley, and J. A. Miller. *Chemkin-II: A Fortran chemical kinetics package for the analysis of gas-phase chemical kinetics*. Tech. rep. Sandia National Labs., Livermore, CA, 1989.

[65] E. Anderson et al. *LAPACK Users’ Guide*. 3rd. Philadelphia, PA: SIAM, 1999.

[66] H. P. Le, J.-L. Cambier, and L. K. Cole. “GPU-based flow simulation with detailed chemical kinetics”. In: *Comput. Phys. Comm.* 184.3 (2013), pp. 596–606. DOI: 10.1016/j.cpc.2012.10.013.

[67] G. Wanner and E. Hairer. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. 2nd ed. Springer-Verlag, Berlin, 1996. DOI: 10.1007/978-3-642-05221-7.

[68] N. Yonkee and J. C. Sutherland. “PoKiTT: Exposing Task and Data Parallelism on Heterogeneous Architectures for Detailed Chemical Kinetics, Transport, and Thermodynamics Calculations”. In: *SIAM J. Sci. Comput.* 38.5 (2016), S264–S281. DOI: 10.1137/15M1026237.

[69] F. Perini, E. Galligani, and R. D. Reitz. "A study of direct and Krylov iterative sparse solver techniques to approach linear scaling of the integration of chemical kinetics with detailed combustion mechanisms". In: *Combust. Flame* 161.5 (2014), pp. 1180–1195. DOI: 10.1016/j.combustflame.2013.11.017.

[70] M. J. McNenly, R. A. Whitesides, and D. L. Flowers. "Faster solvers for large kinetic mechanisms using adaptive preconditioners". In: *Proc. Combust. Inst.* 35.1 (2015), pp. 581–587. DOI: 10.1016/j.proci.2014.05.113.

[71] M. Hochbruck and C. Lubich. "On Krylov Subspace Approximations to the Matrix Exponential Operator". In: *SIAM J. Numer. Anal.* 34.5 (Oct. 1997), pp. 1911–1925. ISSN: 0036-1429. DOI: 10.1137/S0036142995280572.

[72] M. Hochbruck, C. Lubich, and H. Selhofer. "Exponential Integrators for Large Systems of Differential Equations". In: *SIAM J. Sci. Comput.* 19.5 (Sept. 1998), pp. 1552–1574. DOI: 10.1137/S1064827595295337.

[73] F. Bisetti. "Integration of large chemical kinetic mechanisms via exponential methods with Krylov approximations to Jacobian matrix functions". In: *Combust. Theor. Model.* 16.3 (2012), pp. 387–418. DOI: 10.1080/13647830.2011.631032.

[74] M. Falati and G. Hojjati. "Integration of chemical stiff ODEs using exponential propagation method". In: *J. Math. Chem.* 49.10 (2011), pp. 2210–2230. DOI: 10.1007/s10910-011-9881-9.

[75] K. E. Niemeyer and N. J. Curtis. *pyJac v1.0.2*. Jan. 2017. DOI: 10.5281/zenodo.251144.

[76] K. E. Niemeyer, N. J. Curtis, and C. J. Sung. "Initial investigation of pyJac: an analytical Jacobian generator for chemical kinetics". In: *Fall 2015 Meeting of the West. States Sect. Combust. Inst.* Provo, UT, USA, Oct. 2015. DOI: 10.6084/m9.figshare.2075515.

[77] M. J. McNenly, R. A. Whitesides, and D. L. Flowers. "Adaptive Preconditioning Strategies for Integrating Large Kinetic Mechanisms". In: *8th US National Combustion Meeting, Park City, UT*. 2013.

[78] N. J. Curtis and K. Niemeyer. *accelerInt v1.0-beta*. 2017. DOI: 10.5281/zenodo.230256.

[79] M. Hochbruck, A. Ostermann, and J. Schweitzer. "Exponential Rosenbrock-Type Methods". In: *SIAM J. Numer. Anal.* 47.1 (2009), pp. 786–803. DOI: 10.1137/080717717.

[80] E. Banks et al. *SUNDIALS v2.6.2*. http://computation.llnl.gov/projects/sundials-suite-nonlinear-differential-algebraic-equation-solvers/download/sundials-2.6.2.tar.gz. Aug. 2015.

[81] E. Gallopoulos and Y. Saad. “Efficient Solution of Parabolic Equations by Krylov Approximation Methods”. In: *SIAM J. Sci. Stat. Comp.* 13.5 (1992), pp. 1236–1264. DOI: 10.1137/0913071.

[82] L. N. Trefethen, J. A. C. Weideman, and T. Schmelzer. “Talbot quadratures and rational approximations”. In: *BIT Numer. Math.* 46.3 (2006), pp. 653–670. DOI: 10.1007/s10543-006-0077-9.

[83] K. E. Niemeyer. *cf_expm v1.0*. Jan. 2016. DOI: 10.5281/zenodo.44291.

[84] M. Frigo and S. G. Johnson. “The design and implementation of FFTW3”. In: *Proc. IEEE* 93.2 (2005), pp. 216–231. DOI: 10.1109/JPROC.2004.840301.

[85] M. Frigo and S. G. Johnson. *FFTW v3.3.4*. http://www.fftw.org/. Mar. 2014.

[86] G. W. Stewart. *Matrix Algorithms: Volume 1: Basic Decompositions*. Philadelphia: SIAM, 1998. ISBN: 9780898714142. DOI: 10.1137/1.9781611971408.

[87] Y. Saad. “Analysis of Some Krylov Subspace Approximations to the Matrix Exponential Operator”. In: *SIAM J. on Numer. Anal.* 29.1 (1992), pp. 209–228. DOI: 10.1137/0729014.

[88] M. P. Burke et al. “Comprehensive $H_2/O_2$ kinetic model for high-pressure combustion”. In: *Int. J. Chem. Kinet.* 44.7 (2011), pp. 444–474. DOI: 10.1002/kin.20603.

[89] G. P. Smith et al. *GRI-Mech 3.0*. http://www.me.berkeley.edu/gri_mech/. 1999.

[90] J.-Y. Chen. “Stochastic modeling of partially stirred reactors”. In: *Combust. Sci. Technol.* 122.1–6 (1997), pp. 63–94. DOI: 10.1080/00102209708935605.

[91] H. Robertson. *The solution of a set of reaction rate equations*. Ed. by J. Walsh. Academ. Press, 1966, pp. 178–182.

[92] B. N. Datta. *Numerical Linear Algebra and Applications*. Philadelphia, PA: SIAM, 2010.

[93] M. Hochbruck and A. Ostermann. “Exponential integrators”. In: *Acta Numer.* 19 (May 2010), pp. 209–286. ISSN: 1474-0508. DOI: 10.1017/S0962492910000048.

[94] G. Iaccarino et al. “Reynolds averaged simulation of unsteady separated flow”. In: *Int. J. Heat Fluid Flow* 24.2 (2003), pp. 147–156. ISSN: 0142-727. DOI: 10.1016/S0142-727X(02)00210-2.

[95] H. Wang and S. B. Pope. “Large eddy simulation/probability density function modeling of a turbulent jet flame”. In: *Proc. Combust. Inst.* 33.1 (2011), pp. 1319–1330. ISSN: 1540-7489. DOI: 10.1016/j.proci.2010.08.004.

[96] G. Bulat, W. P. Jones, and A. J. Marquis. "Large Eddy Simulation of an industrial gas-turbine combustion chamber using the sub-grid PDF method". In: *Proc. Combust. Inst.* 34.2 (2013), pp. 3155–3164. ISSN: 1540-7489. DOI: 10.1016/j.proci.2012.07.031.

[97] J. A. Ramírez and C. Cortés. "Comparison of Different URANS Schemes for the Simulation of Complex Swirling Flows". In: *Numer. Heat Transf., Part B: Fundam.* 58.2 (2010), pp. 98–120. DOI: 10.1080/10407790.2010.508440.

[98] E. Galloni. "Analyses about parameters that affect cyclic variation in a spark ignition engine". In: *Appl. Therm. Eng.* 29.5–6 (2009), pp. 1131–1137. ISSN: 1359-4311. DOI: 10.1016/j.applthermaleng.2008.06.001.

[99] Intel. *Intel® Xeon® Processor E5-4640 v2 (20M Cache, 2.20 GHz)*. Accessed: 06-06-2016. URL: https://web.archive.org/web/20180801125507/https://ark.intel.com/products/75288/Intel-Xeon-Processor-E5-4640-v2-20M-Cache-2_20-GHz.

[100] Amazon.com. *Buying Choices: NVIDIA Tesla C2075 6GB GDDR5 PCIe Workstation Card*. Accessed: 06-06-2016. URL: http://www.amazon.com/gp/offer-listing/B005OCMZ7A/ref=dp%5C_olp%5C_all%5C_mbc?ie=UTF8%5C&condition=all.

[101] D. A. Schwer et al. "On upgrading the numerics in combustion chemistry codes". In: *Combust. Flame* 128.3 (2002), pp. 270–291. DOI: 10.1016/S0010-2180(01)00352-2.

[102] Amazon.com. *Tesla K40 Graphic Card - 1 GPUs - 745 MHz Core - 12 GB GDDR5 SDRAM*. https://www.amazon.com/Tesla-K40-Graphic-Card-GDDR5/dp/B00KDRRTB8. (Visited on 07/07/2016).

[103] NVIDIA. *NVIDIA Tesla GPUs Datasheet*. URL: https://web.archive.org/web/20160704140929/https://docs.nvidia.com/cuda/cuda-c-programming-guide/#compute-capabilities (visited on 07/07/2016).

[104] C. P. Stone and F. Bisetti. "Comparison of ODE Solvers for Chemical Kinetics and Reactive CFD Applications". In: *AIAA 52nd Aerospace Sciences Meeting (National Harbor, MD)*. 2014. DOI: 10.2514/6.2014-0822.

[105] T. Dijkmans et al. "GPU based simulation of reactive mixtures with detailed chemistry in combination with tabulation and an analytical Jacobian". In: *Comput. Chem. Eng.* 71 (2014), pp. 521–531. DOI: 10.1016/j.compchemeng.2014.09.016.

[106] T. Steihaug and A. Wolfbrandt. "An attempt to avoid exact Jacobian and nonlinear equations in the numerical solution of stiff differential equations". In: *Math. of Comput.* 33.146 (1979), pp. 521–534. DOI: 10.2307/2006293.

[107] B. A. Schmitt and R. Weiner. “Parallel Two-Step W-Methods with Peer Variables”. In: *SIAM J. on Numer. Anal.* 42.1 (2004), pp. 265–282. DOI: 10.1137/S0036142902411057.

[108] J. E. Stone, D. Gohara, and G. Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *IEEE Des. Test* 12.3 (May 2010), pp. 66–73. ISSN: 0740-7475. DOI: 10.1109/MCSE.2010.69.

[109] E. Lindholm et al. “NVIDIA Tesla: A unified graphics and computing architecture”. In: *IEEE micro* 28.2 (2008), pp. 39–55. DOI: 10.1109/MM.2008.31.

[110] N. J. Curtis, K. E. Niemeyer, and C.-J. Sung. “An investigation of GPU-based stiff chemical kinetics integration methods”. In: *Combust. Flame* 179 (2017), pp. 312–324. ISSN: 0010-2180. DOI: 10.1016/j.combustflame.2017.02.005.

[111] C. Safta, H. N. Najm, and O. M. Knio. *TChem - A Software Toolkit for the Analysis of Complex Kinetic Models*. Tech. rep. SAND2011-3282. Sandia National Laboratories, May 2011.

[112] N. J. Curtis and K. E. Niemeyer. *Fileset for testing thread-safety of TChem*. figshare. Jan. 2017. DOI: 10.6084/m9.figshare.4563982.v1.

[113] M. R. Youssefi. “Development of Analytic Tools for Computational Flame Diagnostics”. MA thesis. University of Connecticut, Aug. 2011.

[114] F. Perini, E. Galligani, and R. D. Reitz. “An Analytical Jacobian Approach to Sparse Reaction Kinetics for Computationally Efficient Combustion Modeling with Large Reaction Mechanisms”. In: *Energy Fuels* 26.8 (Aug. 2012), pp. 4804–4822. DOI: 10.1021/ef300747n.

[115] Y. Gao et al. “A dynamic adaptive method for hybrid integration of stiff chemistry”. In: *Combust. Flame* 162.2 (2015), pp. 287–295. ISSN: 0010-2180. DOI: 10.1016/j.combustflame.2014.07.023.

[116] M. A. Hansen and J. C. Sutherland. “On the consistency of state vectors and Jacobian matrices”. In: *Combust. Flame* 193 (2018), pp. 257–271. ISSN: 0010-2180. DOI: j.combustflame.2018.03.017.

[117] M. Bauer, S. Treichler, and A. Aiken. “Singe: Leveraging Warp Specialization for High Performance on GPUs”. In: *SIGPLAN Not.* 49.8 (Feb. 2014), pp. 119–130. ISSN: 0362-1340. DOI: 10.1145/2692916.2555258.

[118] T. F. Lu et al. “Three-dimensional direct numerical simulation of a turbulent lifted hydrogen jet flame in heated coflow: a chemical explosive mode analysis”. In: *J. Fluid Mech.* 652 (2010), pp. 45–64. DOI: 10.1017/S002211201000039X.

[119] J. C. Linford et al. “Automatic Generation of Multicore Chemical Kernels”. In: *IEEE Trans. Parallel Distrib. Syst.* 22.1 (Jan. 2011), pp. 119–131. DOI: 10.1109/TPDS.2010.106.

[120] J. Gray and P. Shenoy. “Rules of thumb in data engineering”. In: *Data Engineering, 2000. Proceedings. 16th International Conference on*. IEEE. 2000, pp. 3–10.

[121] NVIDIA. *CUDA C Programming Guide, version 9.0*. Jan. 2018. URL: https://web.archive.org/web/20181224180607/https://docs.nvidia.com/cuda/archive/9.0/cuda-c-programming-guide/index.html.

[122] S. R. Turns. *An Introduction to Combustion: Concepts and Applications*. Eng. 3rd ed. New York: McGraw-Hill, 2012. ISBN: 9780073380193.

[123] A. Klöckner. “Loo.py: transformation-based code generation for GPUs and CPUs”. In: *Proc. ARRAY '14: ACM SIGPLAN Workshop Libr., Lang., Compil. Array Progr.* Edinburgh, Scotland.: Assoc. Comput. Mach., 2014. DOI: 10.1145/2627373.2627387.

[124] N. J. Curtis and K. E. Niemeyer. *pyJac v1.0.6*. Feb. 2018. DOI: 10.5281/zenodo.1182789.

[125] A. Klöckner et al. “PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation”. In: *Parallel Comput.* 38.3 (2012), pp. 157–174. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.09.001.

[126] D. G. Goodwin, H. K. Moffat, and R. L. Speth. *Cantera: An Object-oriented Software Toolkit for Chemical Kinetics, Thermodynamics, and Transport Processes*. http://www.cantera.org. Version 2.3.0. 2017. DOI: 10.5281/zenodo.170284.

[127] R. J. Hogan. “Fast Reverse-Mode Automatic Differentiation using Expression Templates in C++”. In: *ACM Trans. Math. Software* 40.4 (2014), p. 26. DOI: 10.1145/2560359.

[128] R. J. Hogan. *Adept v1.1*. Available at https://github.com/rjhogan/Adept. June 2015.

[129] P. Jääskeläinen et al. “pocl: A Performance-Portable OpenCL Implementation”. English. In: *Int. J. Parallel Program.* 43.5 (2015), pp. 752–785. ISSN: 0885-7458. DOI: 10.1007/s10766-014-0320-y.

[130] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *Comput. Sci. & Engineering, IEEE* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[131] Travis CI, GmbH. *Travis CI - Test and Deploy Your Code with Confidence*. https://about.travis-ci.com/. 2018. (Visited on 02/12/2018).

[132] Intel® Corporation. *OpenCL™ Drivers and Runtimes for Intel® Architecture*. https://software.intel.com/en-us/articles/opencl-drivers#latest_CPU_runtime. 2018. (Visited on 02/12/2018).

[133] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. ISBN: 0-7695-2102-9. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

[134] MichaelE1000. *Bug report on NVIDIA forums*. NVIDIA Devtalk Forums. Accessed 03-06-18.

[135] H. Wang et al. *USC Mech Version II. High-Temperature Combustion Reaction Model of $H_2/CO/C_1$–$C_4$ Compounds*. http://ignis.usc.edu/USC_Mech_II.htm. May 2007.

[136] S. M. Sarathy et al. “A comprehensive experimental and modeling study of iso-pentanol combustion”. In: *Combust. Flame* 160.12 (2013), pp. 2712–2728. DOI: 10.1016/j.combustflame.2013.06.022.

[137] A. C. Hindmarsh et al. “SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers”. In: *ACM Trans. Math. Softw.* 31.3 (Sept. 2005), pp. 363–396. DOI: 10.1145/1089014.1089020.

[138] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Note: the section on sparse matricies formats. Philadelphia, PA: SIAM, 1994.

[139] M. Babej and P. Jääskeläinen. *Debugging auto vectorizer*. Private Communication. Archived on POCL mailing list. Feb. 2018.

[140] NVIDIA. *Achieved Occupancy*. Achieved Occupancy. Mar. 2018.

[141] Intel® Corporation. *Using Vector Data Types*. Accessed on 02/19/18. 2015. URL: https://web.archive.org/web/20181224031752/https://software.intel.com/en-us/node/540561.

[142] Intel® Corporation. *Vectorizer Knobs*. Accessed on 02/19/18. 2015. URL: https://web.archive.org/web/20181224031901/https://software.intel.com/en-us/node/540560.

[143] G. G. Howes. *Parallel Performance and Optimization*. Slides from Iowa High Performance Computing Summer School, University of Iowa, 08/2010 - Accessed on 02/19/18. URL: https://web.archive.org/web/20170829225747/http://homepage.physics.uiowa.edu/~ghowes/teach/ihpc10/lec/ihpc10Lec_PerformanceHPC10.pdf.

[144] N. J. Curtis. *A minimum working example showing the failure of simple OpenCL code on the NVIDIA Linux x64 Tesla 375.26 Driver*. https://figshare.com/s/03aa9064aa6fe3508d3d. June 2018. DOI: 10.6084/m9.figshare.6533915.

[145] M. Pharr and W. R. Mark. "ispc: A SPMD compiler for high-performance CPU programming". In: *Innovat. Parallel Comput. (InPar), 2012*. 2012, pp. 1–13. DOI: 10.1109/InPar.2012.6339601.

[146] NVIDIA Corporation. *Dense Linear Algebra on GPUs*. Accessed: 03-12-18. 2018. URL: https://developer.nvidia.com/cublas.

[147] clMathLibraries. *clMathLibraries - clBLAS/clSPARSE*. Accessed: 03-12-18. 2018. URL: https://github.com/clMathLibraries.

[148] J. W. Demmel et al. "A supernodal approach to sparse partial pivoting". In: *SIAM J. Matrix Analys. Appl.* 20.3 (1999), pp. 720–755. DOI: 10.1137/S0895479895291765.

[149] N. J. Curtis, K. E. Niemeyer, and C.-J. Sung. "Using SIMD and SIMT vectorization to evaluate sparse chemical kinetic Jacobian matrices and thermochemical source terms". In: *Combustion and Flame* 198 (2018), pp. 186–204. ISSN: 0010-2180. DOI: 10.1016/j.combustflame.2018.09.008.

[150] E. Banks et al. *SUNDIALS v2.7.0*. http://computation.llnl.gov/projects/sundials-suite-nonlinear-differential-algebraic-equation-solvers/download/sundials-2.7.0.tar.gz. Sept. 2016.

[151] H. G. Weller et al. "A tensorial approach to computational continuum mechanics using object-oriented techniques". In: *Comput. Phys.* 12.6 (1998), pp. 620–631. DOI: 10.1063/1.168744.

[152] R. Barlow and J. Frank. "Effects of turbulence on species mass fractions in methane/air jet flames". In: *Twenty-Seventh International Symposium on Combustion* 27.1 (1998), pp. 1087–1095. ISSN: 0082-0784. DOI: 10.1016/S0082-0784(98)80510-9.

[153] R. Barlow et al. "Piloted methane/air jet flames: Transport effects and aspects of scalar structure". In: *Combust. Flame* 143.4 (2005), pp. 433–449. ISSN: 0010-2180. DOI: https://doi.org/10.1016/j.combustflame.2005.08.017.

[154] C. Schneider et al. "Flow field measurements of stable and locally extinguishing hydrocarbon-fuelled jet flames". In: *Combust. Flame* 135.1 (2003), pp. 185–190. ISSN: 0010-2180. DOI: doi.org/10.1016/S0010-2180(03)00150-0.

[155] A. Sandu et al. “Benchmarking stiff ODE solvers for atmospheric chemistry problems II: Rosenbrock solvers”. In: *Atmospheric Environ.* 31.20 (1997), pp. 3459–3472. ISSN: 1352-2310. DOI: 10.1016/S1352-2310(97)83212-8.

[156] H. Zhang and A. Sandu. “FATODE: A Library for Forward, Adjoint, and Tangent Linear Integration of ODEs”. In: *SIAM J. Sci. Comput.* 36.5 (2014), pp. C504–C523. DOI: 10.1137/130912335.

[157] S. P. N. Ernst Hairer and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. ISBN: 978-3-540-56670-0. DOI: 10.1007/978-3-540-78862-1.

[158] P. Kaps, S. W. H. Poon, and T. D. Bui. “Rosenbrock methods for Stiff ODEs: A comparison of Richardson extrapolation and embedding technique”. In: *Comput.* 34.1 (Mar. 1985), pp. 17–40. ISSN: 1436-5057. DOI: 10.1007/BF02242171.

[159] L. F. Shampine. “Implementation of Rosenbrock Methods”. In: *ACM Trans. Math. Softw.* 8.2 (June 1982), pp. 93–113. ISSN: 0098-3500. DOI: 10.1145/355993.355994.

[160] B. E. Launder and B. I. Sharma. “Application of the energy-dissipation model of turbulence to the calculation of flow near a spinning disc”. In: *Lett. Heat Mass Transf.* 1 (Dec. 1974), pp. 131–137.

[161] W. Jones and B. Launder. “The prediction of laminarization with a two-equation model of turbulence”. In: *Int. J. Heat Mass Transf.* 15.2 (1972), pp. 301–314. ISSN: 0017-9310. DOI: 10.1016/0017-9310(72)90076-2.

[162] D. A. Lysenko, I. S. Ertesvåg, and K. E. Rian. “Numerical Simulation of Non-premixed Turbulent Combustion Using the Eddy Dissipation Concept and Comparing with the Steady Laminar Flamelet Model”. In: *Flow, Turbul. Combust.* 93.4 (Dec. 2014), pp. 577–605. ISSN: 1573-1987. DOI: 10.1007/s10494-014-9551-7.

[163] C. Fureby. “Large eddy simulation modeling of combustion for propulsion applications”. In: *Philos. Trans. Royal Soc. Lond. A: Math., Phys. Eng. Sci.* 367.1899 (2009), pp. 2957–2969. ISSN: 1364-503X. DOI: 10.1098/rsta.2008.0271.

[164] The OpenFOAM Foundation. *OpenFOAM v6 User Guide*. [Online; accessed 12/02/18]. URL: https://cfd.direct/openfoam/user-guide.

[165] M. P. Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. Knoxville, TN, USA, 1994.

[166] H. Weller et al. “Application of a flame-wrinkling LES combustion model to a turbulent mixing layer”. In: vol. 27. 1. 1998, pp. 899–907. DOI: 10.1016/S0082-0784(98)80487-6.

[167] B. F. Magnussen. “The Eddy Dissipation Concept—A Bridge Between Science and Technology”. In: *ECCOMAS thematic conference on computational combustion*. 2005, pp. 21–24.

[168] M. Bösenhofer et al. “The Eddy Dissipation Concept—Analysis of Different Fine Structure Treatments for Classical Combustion”. In: *Energies* 11.7 (2018). ISSN: 1996-1073. DOI: 10.3390/en11071902.

[169] M. J. Evans, P. R. Medwell, and Z. F. Tian. “Modeling Lifted Jet Flames in a Heated Coflow Using an Optimized Eddy Dissipation Concept Model”. In: *Combust. Sci. Tech.* 187.7 (2015), pp. 1093–1109. DOI: 10.1080/00102202.2014.1002836.

[170] Z. Li et al. “Comprehensive numerical study of the Adelaide Jet in Hot-Coflow burner by means of RANS and detailed chemistry”. In: *Energy* 139 (2017), pp. 555–570. ISSN: 0360-5442. DOI: 10.1016/j.energy.2017.07.132.

[171] B. F. Magnussen. “Modeling of NOx and soot formation by the eddy dissipation concept”. In: *International Flame Research Foundation First Topic Oriented Technical Meeting*. 1989, pp. 17–19.

[172] N. J. Curtis and K. E. Niemeyer. *ch4_pasr_data.bin*. Figshare. Oct. 2016. DOI: 10.6084/m9.figshare.4007418.v2.

[173] D. G. Goodwin, H. K. Moffat, and R. L. Speth. *Cantera: An Object-oriented Software Toolkit for Chemical Kinetics, Thermodynamics, and Transport Processes*. http://www.cantera.org. Version 2.4.0. 2018. DOI: 10.5281/zenodo.1174508.

[174] The OpenFOAM Foundation. *OpenFOAM | Free CFD Software*. [Online; accessed 12/02/18]. URL: https://openfoam.org/.

[175] E. Gabriel et al. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Proceedings, 11th European PVM/MPI Users’ Group Meeting*. Budapest, Hungary, Sept. 2004, pp. 97–104.

[176] R. M. Stallman and GCC Developer Community. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Paramount, CA: CreateSpace, 2009. ISBN: 144141276X, 9781441412768.

[177] D. Skinner. *Performance monitoring of parallel scientific applications*. Tech. rep. May 2005. DOI: 10.2172/881368.

[178] A. Sjunnesson, S. Olovsson, and B. Sjoblom. “Validation rig - A tool for flame studies”. In: *International Symposium on Air Breathing Engines, 10th, Nottingham, England*. 1991, pp. 385–393.

[179] A. Sjunnesson, C. Nelsson, and E. Max. “LDA measurements of velocities and turbulence in a bluff body stabilized flame”. In: *Laser Anemometry* 3 (1991), pp. 83–90.

[180] J. Kodavasal et al. “Development of a Stiffness-Based Chemistry Load Balancing Scheme, and Optimization of Input/Output and Communication, to Enable Massively Parallel High-Fidelity Internal Combustion Engine Simulations”. In: *J. Energy Resour. Technol.* 138.5 (2016), p. 052203.

[181] A. Alferman. “Evaluating Stiffness Metrics for Predicting the Cost of Chemical Kinetics Integration”. MA thesis. Oregon State University, Aug. 2018.

[182] T. A. Davis. *Direct methods for sparse linear systems*. Vol. 2. Siam, 2006.

[183] NVIDIA Corporation. *cuSPARSE | NVIDIA Developer*. Accessed: 03-12-18. 2018. URL: https://developer.nvidia.com/cusparse.

[184] D. B. Kirk and W. H. Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2016.

[185] A. Rahunanthan and D. Stanescu. “High-order W-methods”. In: *J. Comput. Appl. Math.* 233.8 (2010), pp. 1798–1811. DOI: 10.1016/j.cam.2009.09.017.

[186] B. A. Schmitt, R. Weiner, and H. Podhaisky. “Multi-Implicit Peer Two-Step W-Methods for Parallel Time Integration”. In: *BIT Numer. Math.* 45.1 (2005), pp. 197–217. DOI: 10.1007/s10543-005-2635-y.

[187] H. Podhaisky, R. Weiner, and B. Schmitt. “Rosenbrock-type "Peer" two-step methods”. In: *Appl. Numer. Math.* 53.2 (2005). Tenth Seminar on Numerical Solution of Differential and Differntial-Algebraic Euqations (NUMDIFF-10), pp. 409–420. ISSN: 0168-9274. DOI: 10.1016/j.apnum.2004.08.021.

[188] S. M. Correa. “Turbulence-Chemistry Interactions in the Intermediate Regime of Premixed Combustion”. In: *Combust. Flame* 93.1–2 (1993), pp. 41–60. DOI: 10.1016/0010-2180(93)90083-F.

[189] A. Bhave and M. Kraft. “Partially Stirred Reactor Model: Analytical Solutions and Numerical Convergence Study of a PDF/Monte Carlo Method”. In: *SIAM J. Sci. Comput.* 25.5 (2004), pp. 1798–1823. DOI: 10.1137/S1064827502411328.

[190] Z. Ren and S. B. Pope. “An investigation of the performance of turbulent mixing models”. In: *Combust. Flame* 136.1-2 (2004), pp. 208–216. DOI: 10.1016/j.combustflame.2003.09.014.

[191] A. Meurer et al. “SymPy: symbolic computing in Python”. In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103.

[192] S. Gordon and B. J. McBride. *Computer program for calculation of complex chemical equilibrium compositions and applications*. Vol. 1. National Aeronautics, Space Administration, Office of Management, Scientific, and Technical Information Program, 1994.

[193] F. A. Lindemann et al. "Discussion on "the radiation theory of chemical action"". In: *Trans. Faraday Soc.* 17 (1922), pp. 598–606.

[194] R. G. Gilbert, K. Luther, and J. Troe. "Theory of Thermal Unimolecular Reactions in the Fall-off Range. II. Weak Collision Rate Constants". In: *Ber. Bunsenges. Phys. Chem.* 87.2 (Feb. 1983), pp. 169–177.

[195] P. H. Stewart, C. W. Larson, and D. M. Golden. "Pressure and temperature dependence of reactions proceeding via a bound complex. 2. Application to $2CH_3 \rightarrow C_2H_5 + H$". In: *Combust. Flame* 75.1 (1989), pp. 25–31.

[196] Reaction Design: San Diego. *CHEMKIN-PRO 15113*. 2012.

[197] P. K. Venkatesh et al. "Chebyshev Expansions and Sensitivity Analysis for Approximating the Temperature- and Pressure-Dependence of Chemically-Activated Reactions". In: *Rev. Chem. Eng.* 13.1 (Mar. 1997), pp. 1–67.

[198] P. K. Venkatesh et al. "Parameterization of pressure- and temperature-dependent kinetics in multiple well reactions". In: *AICHe J.* 43.5 (May 1997), pp. 1331–1340.

[199] P. K. Venkatesh. "Damped Pseudospectral Functional Forms of the Falloff Behavior of Unimolecular Reactions". In: *J. Phys. Chem. A* 104.2 (2000), pp. 280–287.

[200] N. J. Curtis and K. E. Niemeyer. ***pyJac*** *v2.0.0-beta.0*. June 2018. DOI: 10.5281/zenodo.1289979.

[201] N. J. Curtis, K. E. Niemeyer, and C.-J. Sung. *Data, plotting scripts, and figures for "Using SIMD and SIMT vectorization to evaluate sparse chemical kinetic Jacobian matrices and thermochemical source terms"*. June 2018. DOI: 10.6084/m9.figshare.6534146.

# Appendix A

# Supplemental: Partially stirred reactor implementation

Here, we describe our partially stirred reactor (PaSR) implementation for completeness, based on prior descriptions [37, 38, 90, 188–190]. The reactor model consists of an even number $N_{\text{part}}$ of particles, each with a time-varying composition $\phi(t)$. Here we use mixture enthalpy and species mass fractions to describe the state of a particle:

$$\phi = \left\{h, Y_1, Y_2, \ldots, Y_{N_{\text{sp}}}\right\}^{\intercal} \ . \tag{A.1}$$

At discrete time steps of size $\Delta t$, events including inflow, outflow, and pairing cause certain particles to change composition; between these time steps, mixing and reaction fractional steps separated by step size $\Delta t_{\text{sub}}$ evolve the composition of all particles.

Inflow and outflow events at the discrete time steps comprise the inflow stream compositions replacing the compositions of $N_{\text{part}}\Delta t/\tau_{\text{res}}$ randomly selected particles, where $\tau_{\text{res}}$ is the residence time. For premixed combustion cases, the inflow streams consist of a fresh fuel/air mixture stream at a specified temperature and equivalence ratio, and a pilot stream consisting of the adiabatic equilibrium products of the fresh mixture stream, with the mass flow rates of these two streams in a ratio of $0.95 : 0.05$. Non-premixed cases consist of three inflow streams: air, fuel, and a pilot consisting of the adiabatic equilibrium products of a stoichiometric fuel/air mixture at the same unburned temperature as the first two streams; the mass flow rates of these streams occur in a ratio of $0.85 : 0.05 : 0.1$. Following inflow/outflow (for both premixed and non-premixed cases), $\frac{1}{2}N_{\text{part}}\Delta t/\tau_{\text{pair}}$ pairs of particles (not including the inflowing particles) are randomly selected for pairing and then randomly shuffled with the inflowing particles to exchange partners, where $\tau_{\text{pair}}$ is the pairing timescale.

Although multiple mixing models exist [190], the current mixing fractional step consists of a pair of particles $\phi^p$ and $\phi^q$ exchanging compositional information and evolving by

$$\frac{\mathrm{d}\phi^p}{\mathrm{d}t} = -\frac{\phi^p - \phi^q}{\tau_{\mathrm{mix}}} \quad \text{and} \tag{A.2}$$

$$\frac{\mathrm{d}\phi^q}{\mathrm{d}t} = -\frac{\phi^q - \phi^p}{\tau_{\mathrm{mix}}} \ , \tag{A.3}$$

where $\tau_{\mathrm{mix}}$ is a characteristic mixing timescale. The analytical solution to this system of equations determines the particle compositions $\phi^p$ and $\phi^q$ after a mixture fractional step:

$$\phi^p = \phi_0^p - \alpha \ , \tag{A.4}$$

$$\phi^q = \phi_0^q + \alpha \ , \ \text{and} \tag{A.5}$$

$$\alpha = \frac{\phi_0^p - \phi_0^q}{2}\left[1 - \exp\left(\frac{-2\Delta t_{\mathrm{sub}}}{\tau_{\mathrm{mix}}}\right)\right] \ , \tag{A.6}$$

where $\phi_0^p$ and $\phi_0^q$ are the particle compositions at the beginning of the mixture fractional step and $\Delta t_{\mathrm{sub}}$ is the mixing sub-time-step size. The reaction fractional step consists of the enthalpy evolving by

$$\frac{\mathrm{d}h}{\mathrm{d}t} = \frac{-1}{\rho}\sum_{k=1}^{N_{\mathrm{sp}}} h_k W_k \dot{\omega}_k \tag{A.7}$$

and the species mass fractions evolving according to:

$$\frac{\mathrm{d}Y_k}{\mathrm{d}t} = \frac{W_k}{\rho}\dot{\omega}_k \quad k = 1, \ldots, N_{\mathrm{sp}} - 1 \ , \tag{A.8}$$

However, in practice our implementation handles the reaction fractional step by advancing in time a Cantera [126] `ReactorNet` that contains a `IdealGasConstPressureReactor` object, rather than integrating the above equations directly.

The time integration scheme implemented in our approach determines the discrete time step between inflow/outflow and pairing events $\Delta t$ and the sub-time step size $\Delta t_{\mathrm{sub}}$ separating mixing/reaction fractional steps, both held constant in the current implementation, via

$$\Delta t = 0.1 \ \min\left(\tau_{\mathrm{res}}, \tau_{\mathrm{pair}}\right) \ \text{and} \tag{A.9}$$

$$\Delta t_{\mathrm{sub}} = 0.04\,\tau_{\mathrm{mix}} \ , \tag{A.10}$$

adopted from Pope [37].

Figures A.1 and A.2 demonstrate sample results from premixed PaSR combustion of methane/air, using GRI-Mech 3.0 [89]; Fig. A.1 shows the mean temperature evolution over time, while Fig. A.2 shows the temperature distribution among all particles. Although a large number

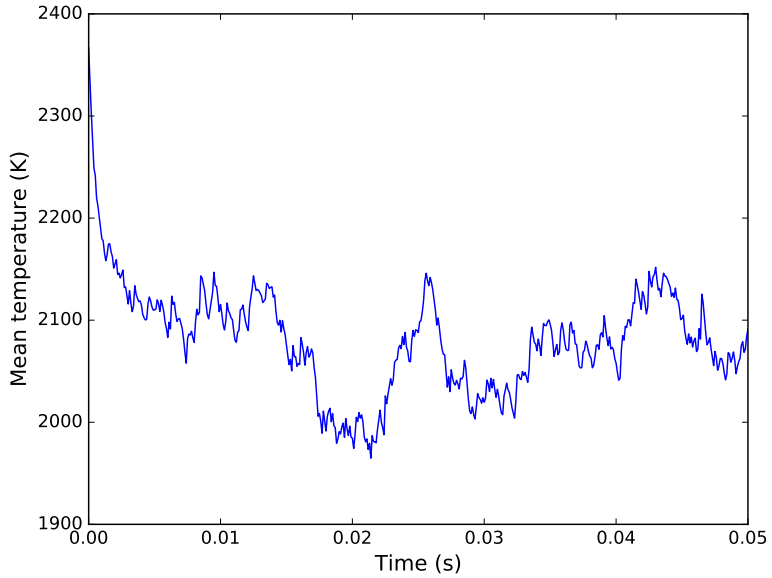



Figure A.1: Mean temperature of premixed PaSR combustion for stoichiometric methane/air with an unburned temperature of 600 K and at 1 atm, $\tau_{\text{res}} = 5\,\text{ms}$, $\tau_{\text{mix}} = \tau_{\text{pair}} = 1\,\text{ms}$, and using 100 particles. Data, plotting scripts, and the figure file are available under CC-BY [60].

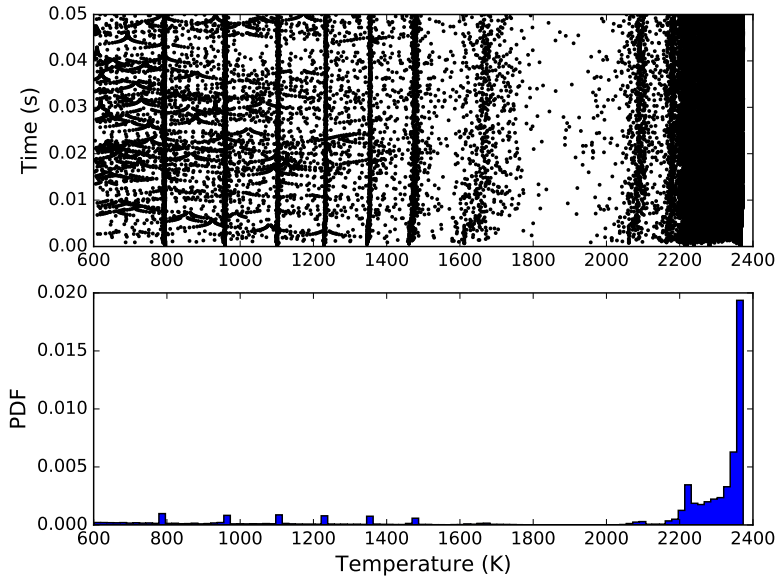



Figure A.2: Scatterplot of temperature over time (top) and probability density function (PDF) of temperature (bottom) of premixed PaSR combustion for stoichiometric methane/air with an unburned temperature of 600 K and at 1 atm, $\tau_{\text{res}} = 5\,\text{ms}$, $\tau_{\text{mix}} = \tau_{\text{pair}} = 1\,\text{ms}$, and using 100 particles. Data, plotting scripts, and the figure file are available under CC-BY [60].

of particles reside near the equilibrium temperature of 2367 K, the wide distribution in particle states is evident.

# Appendix B

# Supplemental Materials for "GPU-based stiff chemical kinetics integration methods"

The results for Chapter 2 of this work were obtained using `accelerInt` v1.0-beta [78]. The most recent version of `accelerInt` can be found at its GitHub repository https://github.com/SLACKHA/accelerInt. All figures as well as the data and plotting scripts necessary to reproduce them, are available openly under the CC-BY license [60].
In addition, [60] includes unscaled plots of integrator runtimes and characterizations of the partially stirred reactor data for this work.

# Appendix C

# Supplemental: A System of Equations and Derivation of Sparse Constant-Pressure/ Constant-Volume Chemical Kinetic Jacobians for `pyJac`

## C.1 Introduction

This appendix is provided as a reference guide to the equations evaluated by `pyJac` v2.0, a code-generation platform for the evaluatation of chemical source terms and analytical chemical kinetic Jacobians of constant-pressure/constant-volume, fixed-mass, adiabatic reactors. Note that equations specific to the constant-pressure or constant-volume derivation will be marked with CONP or CONV respectively. The derivations in this document are based upon the output of the script derivations.py in the GitHub repository for [149]. This script depends on the symbolic mathematics library SymPy [191], and was tested with versions 1.0 and 1.1.1

## C.2 Governing equations

### C.2.1 State variables

The state vector for this derivation consists of the temperature, a thermodynamic state parameter (pressure or volume) and the number of moles of all species except the last in the model:

$$\Phi = \left\{T, V, n_1, n_2 \dots n_{N_{\text{sp}}-1}\right\} \qquad \text{for CONP,} \tag{C.1a}$$

$$\Phi = \left\{T, P, n_1, n_2 \dots n_{N_{\text{sp}}-1}\right\} \qquad \text{for CONV,} \tag{C.1b}$$

where $T$ is the temperature, $V$ and $P$ the volume and pressure, respectively, and $n_j$ the number of moles of the $j$th species in the model (containing $N_{\text{sp}}$ total species).
From the ideal gas law, the number of moles of the final species is determined as:

$$\begin{aligned} n &= \frac{VP}{T\mathcal{R}} = \sum_{i=1}^{N_{\text{sp}}} n_i \ , \\ n_{N_{\text{sp}}} &= \frac{VP}{T\mathcal{R}} - \sum_{i=1}^{N_{\text{sp}}-1} n_i \ , \end{aligned} \tag{C.2}$$

where $\mathcal{R}$ is the universal gas constant in molar units.

### C.2.2 Thermochemical source terms

The evolution of the thermochemical state variables of this system is described by a set of ordinary-differential equations:

$$f = \frac{\mathrm{d}\Phi}{\mathrm{d}t} = \left\{\frac{\mathrm{d}T}{\mathrm{d}t}, \frac{\mathrm{d}V}{\mathrm{d}t}, \frac{\mathrm{d}n_1}{\mathrm{d}t}, \frac{\mathrm{d}n_2}{\mathrm{d}t} \dots \frac{\mathrm{d}n_{N_{\text{sp}}-1}}{\mathrm{d}t}\right\} \qquad \text{for CONP,} \tag{C.3a}$$

$$f = \frac{\mathrm{d}\Phi}{\mathrm{d}t} = \left\{\frac{\mathrm{d}T}{\mathrm{d}t}, \frac{\mathrm{d}P}{\mathrm{d}t}, \frac{\mathrm{d}n_1}{\mathrm{d}t}, \frac{\mathrm{d}n_2}{\mathrm{d}t} \dots \frac{\mathrm{d}n_{N_{\text{sp}}-1}}{\mathrm{d}t}\right\} \qquad \text{for CONV.} \tag{C.3b}$$

For both CONP and CONV, the molar source terms are [122]:

$$\frac{\mathrm{d}n_k}{\mathrm{d}t} = V\dot{\omega}_k \qquad k = 1, \dots, N_{\text{sp}} - 1, \tag{C.4}$$

where $\dot{\omega}_k$ is the $kth$ species' overall production rate, and the temperature production rate [122] is:

$$\frac{\mathrm{d}T}{\mathrm{d}t} = -\frac{\sum_{k=1}^{N_{\text{sp}}} H_k\dot{\omega}_k}{\sum_{k=1}^{N_{\text{sp}}}[C]_k C_{p,k}} \qquad \text{for CONP,} \tag{C.5a}$$

$$\frac{\mathrm{d}T}{\mathrm{d}t} = -\frac{\sum_{k=1}^{N_{\text{sp}}} U_k\dot{\omega}_k}{\sum_{k=1}^{N_{\text{sp}}}[C]_k C_{v,k}} \qquad \text{for CONV,} \tag{C.5b}$$

where $H_k$, $U_k$, $C_{p,k}$ and $C_{v,k}$ are the enthalpy, internal energy, constant-pressure and constant-volume specific heats of species $k$ in molar units, respectively, while $[C]_k$ is the concentration, given by:

$$[C]_k = \frac{n_k}{V} \,. \tag{C.6}$$

From the ideal gas law, source terms for the volume and pressure variables may derived:

$$\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{\mathcal{R}}{P}\left(T\frac{\mathrm{d}n}{\mathrm{d}t} + \frac{\mathrm{d}T}{\mathrm{d}t}n\right) \qquad \text{for CONP,} \tag{C.7a}$$

$$\frac{\mathrm{d}P}{\mathrm{d}t} = \frac{\mathcal{R}}{V}\left(T\frac{\mathrm{d}n}{\mathrm{d}t} + \frac{\mathrm{d}T}{\mathrm{d}t}n\right) \qquad \text{for CONV.} \tag{C.7b}$$

To determine $\frac{\mathrm{d}n}{\mathrm{d}t}$, conservation of mass is invoked:

$$\frac{\mathrm{d}m}{\mathrm{d}t} = 0 = \sum_{k=1}^{N_{\mathrm{sp}}} W_k \frac{\mathrm{d}n_k}{\mathrm{d}t}, \tag{C.8}$$

where $W_k$ is the molecular weight of the $k$th species. From Eq. (C.8), the source term of the last species may be written in terms of the species included in the state vector:

$$\frac{\mathrm{d}n_{N_{\mathrm{sp}}}}{\mathrm{d}t} = -\sum_{k=1}^{N_{\mathrm{sp}}-1} \frac{W_k}{W_{N_{\mathrm{sp}}}} \frac{\mathrm{d}n_k}{\mathrm{d}t} \tag{C.9}$$

where $W_{N_{\mathrm{sp}}}$ is the molecular weight of species $N_{\mathrm{sp}}$. Using Eq. (C.9), the total molar rate of change can be determined:

$$\begin{aligned} \frac{\mathrm{d}n}{\mathrm{d}t} &= \sum_{k=1}^{N_{\mathrm{sp}}} \frac{\mathrm{d}n_k}{\mathrm{d}t}, \\ \frac{\mathrm{d}n}{\mathrm{d}t} &= \sum_{k=1}^{N_{\mathrm{sp}}-1} \left(1 - \frac{W_k}{W_{N_{\mathrm{sp}}}}\right) \frac{\mathrm{d}n_k}{\mathrm{d}t} \end{aligned} \tag{C.10}$$

Combining Eqs. (C.4), (C.7) and (C.10) gives the final form of the pressure and volume terms:

$$\frac{\mathrm{d}V}{\mathrm{d}t} = V\left(\frac{T\mathcal{R}}{P}\sum_{k=1}^{N_{\mathrm{sp}}-1}\left(1 - \frac{W_k}{W_{N_{\mathrm{sp}}}}\right)\dot{\omega}_k + \frac{1}{T}\frac{\mathrm{d}T}{\mathrm{d}t}\right) \qquad \text{for CONP,} \tag{C.11a}$$

$$\frac{\mathrm{d}P}{\mathrm{d}t} = T\mathcal{R}\sum_{k=1}^{N_{\mathrm{sp}}-1}\left(1 - \frac{W_k}{W_{N_{\mathrm{sp}}}}\right)\dot{\omega}_k + \frac{P}{T}\frac{\mathrm{d}T}{\mathrm{d}t} \qquad \text{for CONV.} \tag{C.11b}$$

Additionally, the concentration and moles of the last species in the model (which is not a state variable) may be expanded in terms of the state variables of the system:

$$[C]_{N_{\mathrm{sp}}} = [C] - \sum_{k=1}^{N_{\mathrm{sp}}-1} [C]_k \,, \tag{C.12}$$

where $[C]$ is the total concentration:

$$[C] = \frac{P}{T\mathcal{R}} \ . \tag{C.13}$$

Using Eqs. (C.4), (C.9) and (C.12), the temperature source terms may be expanded to contain only state variables (i.e., by removal of the last species' source term and overall production rate):

$$\frac{\mathrm{d}T}{\mathrm{d}t} = -\frac{\sum_{k=1}^{N_{\mathrm{sp}}-1}\left(H_k - \frac{W_k H_{N_{\mathrm{sp}}}}{W_{N_{\mathrm{sp}}}}\right)\dot{\omega}_k}{[C]C_{p,N_{\mathrm{sp}}} + \sum_{k=1}^{N_{\mathrm{sp}}-1}\left(C_{p,k} - C_{p,N_{\mathrm{sp}}}\right)[C]_k} \qquad \text{for CONP,} \tag{C.14a}$$

$$\frac{\mathrm{d}T}{\mathrm{d}t} = -\frac{\sum_{k=1}^{N_{\mathrm{sp}}-1}\left(U_k - \frac{W_k U_{N_{\mathrm{sp}}}}{W_{N_{\mathrm{sp}}}}\right)\dot{\omega}_k}{[C]C_{v,N_{\mathrm{sp}}} + \sum_{k=1}^{N_{\mathrm{sp}}-1}\left(C_{v,k} - C_{v,N_{\mathrm{sp}}}\right)[C]_k} \qquad \text{for CONV.} \tag{C.14b}$$

This form of the temperature source term will be used for derivation of Jacobian terms, but Eq. (C.5) will often be used as well due to its compactness.

### C.2.3 Thermal properties

The standard-state thermodynamic properties (in molar units) for a gaseous species $k$ is defined using the standard seven-coefficient polynomial of Gordon and McBride [192]:

$$\frac{C_{p,k}^{\circ}}{\mathcal{R}} = a_{0,k} + T\left(a_{1,k} + T\left(a_{2,k} + T\left(a_{3,k} + a_{4,k}T\right)\right)\right) \ , \tag{C.15}$$

$$\frac{H_k^{\circ}}{\mathcal{R}} = T\left(a_{0,k} + T\left(\frac{a_{1,k}}{2} + T\left(\frac{a_{2,k}}{3} + T\left(\frac{a_{3,k}}{4} + \frac{a_{4,k}}{5}T\right)\right)\right)\right) + a_{5,k} \ , \tag{C.16}$$

$$\frac{S_k^{\circ}}{\mathcal{R}} = a_{0,k}\ln T + T\left(a_{1,k} + T\left(\frac{a_{2,k}}{2} + T\left(\frac{a_{3,k}}{3} + \frac{a_{4,k}}{4}T\right)\right)\right) + a_{6,k} \ , \tag{C.17}$$

where $C_{p,k}$ and $H_k$ are as described previously, $S_k$ is the entropy in molar units, and the $^{\circ}$ indicates a standard-state property at one atmosphere (equivalent to the property at any pressure for calorically perfect gases).

### C.2.4 Reaction rate expressions

The net species production rate for species $k$ is defined as:

$$\dot{\omega}_k = \sum_{i=1}^{N_{\mathrm{reac}}} \nu_{k,i} q_i \ , \tag{C.18}$$

where $N_{\mathrm{reac}}$ is the number of reactions in the chemical kinetic model, $\nu_{k,i}$ the overall stoichiometric coefficient of species $k$ in reaction $i$ and $q_i$ the net rate-of-progress for reaction $i$:

$$\nu_{k,i} = \nu''_{k,i} - \nu'_{k,i} \ , \tag{C.19}$$

$$q_i = R_i c_i \ , \tag{C.20}$$

with $\nu''_{k,i}$ and $\nu'_{k,i}$ the product and reactant stoichiometric cofficients (respectively) of species $k$ in reaction $i$. The base rate-of-progress for the $i$th reversible reaction $R_i$ is given by:

$$R_i = R_{f,i} - R_{r,i} \ , \tag{C.21}$$

$$R_{f,i} = k_{f,i} \prod_{k=1}^{N_{\rm sp}} [C]_k^{\nu'_{k,i}} \ , \tag{C.22}$$

$$R_{r,i} = k_{r,i} \prod_{k=1}^{N_{\rm sp}} [C]_k^{\nu''_{k,i}} \ , \tag{C.23}$$

where $k_{f,i}$ and $k_{r,i}$ are the forward and reverse reaction coefficients (respectively) for the $i$th reaction, and the third-body/pressure modification $c_i$ is given by:

$$c_i = \begin{cases} 1 & \text{for elementary reactions,} \\ [X]_i & \text{for third-body enhanced reactions,} \\ \dfrac{P_{r,i}}{1+P_{r,i}} F_i & \text{for unimolecular/recombination falloff reactions, and} \\ \dfrac{1}{1+P_{r,i}} F_i & \text{for chemically-activated bimolecular reactions,} \end{cases} \tag{C.24}$$

where for the $i$th reaction $[X]_i$ is the third-body concentration, $P_{r,i}$ is the reduced pressure, and $F_i$ is the falloff blending factor. These terms are defined in the following sections.

The forward reaction rate coefficient $k_{f,i}$ is given by the three-parameter Arrhenius expression:

$$k_{f,i} = A_i T^{\beta_i} \exp\left(-\frac{E_{ai}}{T\mathcal{R}}\right) \ , \tag{C.25}$$

where $A_i$ is the pre-exponential factor, $\beta_i$ is the temperature exponent, and $T_{a,i}$ is the activation temperature given by $T_{a,i} = E_{a,i}/\mathcal{R}$.

As given by Lu and Law [16], depending on the value of the Arrhenius parameters, $k_{f,i}$ can be calculated in different ways to minimize the computational cost:

$$k_{f,i} = \begin{cases} A_i & \text{if } \beta = 0 \text{ and } T_{a,i} = 0 \ , \\ \exp\left(\log A_i + \beta_i \log T\right) & \text{if } \beta_i \neq 0 \text{ and } T_{a,i} = 0 \ , \\ \exp\left(\log A_i + \beta_i \log T - T_{a,i}/T\right) & \text{if } \beta_i \neq 0 \text{ and } T_{a,i} \neq 0 \ , \\ \exp\left(\log A_i - T_{a,i}/T\right) & \text{if } \beta_i = 0 \text{ and } T_{a,i} \neq 0 \ , \text{ and} \\ A_i \prod^{\beta_i} T & \text{if } T_{a,i} = 0 \text{ and } \beta_i \in \mathbb{Z} \ , \end{cases} \tag{C.26}$$

where $\mathbb{Z}$ is the set of integers; the extent of this specialization can be controlled when generating code via `pyjac`.

### C.2.5 Reverse rate coefficient

By definition, the reverse rate coefficient, $k_{r,i}$, of irreversible reactions is zero, while reversible reactions may have non-zero $k_{r,i}$. Note that in `pyJac`, reversible reactions with explicit reverse Arrhenius parameters are split into two irreversible reactions; this simplifies calculation inside the generated code, and eases comparison to Cantera [173] which applies the same transformation. For reversible reactions without an explicit parameterization, the reverse rate coefficient is calculated from ratio of the forward rate coefficient and the equilibrium constant:

$$k_{r,i} = \frac{k_{f,i}}{K_{c,i}} \,, \tag{C.27}$$

$$K_{c,i} = K_{p,i} \left(\frac{P_{atm}}{T\mathcal{R}}\right)^{\sum_{k=1}^{N_{\text{sp}}} \nu_{k,i}} \,, \text{ and} \tag{C.28}$$

$$K_{p,i} = \exp\left(\frac{\Delta S_i^\circ}{\mathcal{R}} - \frac{\Delta H_i^\circ}{\mathcal{R}T}\right) = \exp\left(\sum_{k=1}^{N_{\text{sp}}} \nu_{ki} \left(\frac{S_k^\circ}{\mathcal{R}} - \frac{H_k^\circ}{\mathcal{R}T}\right)\right) \,, \tag{C.29}$$

where $P_{atm}$ is the pressure of one standard atmosphere in appropriate units.
By combining Eqs. (C.28) and (C.29), we obtain:

$$K_{c,i} = \left(\left(\frac{P_{atm}}{\mathcal{R}}\right)^{\sum_{k=1}^{N_{\text{sp}}} \nu_{k,i}}\right) \exp\left(\sum_{k=1}^{N_{\text{sp}}} \nu_{k,i} B_k\right) \,, \tag{C.30}$$

where from Eqs. (C.16) and (C.17), $B_k$ is:

$$\begin{aligned} B_k &= \frac{S_k^\circ}{\mathcal{R}} - \frac{H_k^\circ}{\mathcal{R}T} - \ln T \,, \\ B_k &= T\left(T\left(T\left(\frac{Ta_{k,4}}{20} + \frac{a_{k,3}}{12}\right) + \frac{a_{k,2}}{6}\right) + \frac{a_{k,1}}{2}\right) \\ &\quad + (a_{k,0} - 1)\log(T) - a_{k,0} + a_{k,6} - \frac{a_{k,5}}{T} \,. \end{aligned} \tag{C.31}$$

### C.2.6 Third-body effects

For a reaction enhanced (or diminished) by the presence of a third body, the reaction rate is modified by the third-body concentration $[X]_i$ given by:

$$[X]_i = \sum_{k=1}^{N_{\text{sp}}} \alpha_{k,i}[C]_k \,, \tag{C.32}$$

where $\alpha_{k,i}$ is the third-body efficiency of species $k$ in the $i$th reaction. For a default third-body efficiency of $\alpha_{k,i} = 1$, this may be rearranged to the compact-storage form:

$$[X]_i = [C] + \sum_{k=1}^{N_{\text{sp}}} (\alpha_{k,i} - 1)\,[C]_k \ , \tag{C.33}$$

where only species with non-default third-body efficiencies must be stored in the model.

Expanding the concentration of the last species gives:

$$[X]_i = [C]\alpha_{N_{\text{sp}},i} + \sum_{k=1}^{N_{\text{sp}}-1} \left(-\alpha_{N_{\text{sp}},i} + \alpha_{k,i}\right)[C]_k \ . \tag{C.34}$$

This form will be denoted as the mixture-based (or **mix** for short) third-body efficiency.

If all species in the mixture contribute equally as third bodies, i.e., $\alpha_{k,i} = 1$ for all species, then:

$$[X]_i = [C] = \frac{P}{\mathcal{R}T} \ . \tag{C.35}$$

This form will be called the **unity**-based third-body efficiency.

In addition, a single species may act as the third body in which case

$$[X]_i = [C]_m \ , \tag{C.36}$$

where the $m$th species is the third body. For evaluation of Jacobian entries, the expanded form:

$$[X]_i = \left([C] - \sum_{k=1}^{N_{\text{sp}}-1} [C]_k\right)\delta_{N_{\text{sp}},m} + \left(-\delta_{N_{\text{sp}},m} + 1\right)[C]_m \tag{C.37}$$

will be used, where the Kronecker delta $\delta_{N_{\text{sp}},m}$ is unity if and only if the third body species $m$ is the last species in the model. Equation (C.37) will be referred to as the **species**-based third-body efficiency.

### C.2.7 Falloff and chemically-activated reactions

Unlike elementary and third-body reactions, falloff/chemically-activated reactions exhibit a pressure dependence described as a blending of rates at low- and high-pressure limits; thus, the rate coefficients depend on a mixture of low-pressure ($k_{0,i}$) and high-pressure-limit ($k_{\infty,i}$) coefficients, each with corresponding Arrhenius parameters and expressed using Eq. (C.25). It is noted that when the forward rate coefficient is used—e.g., as in Eq. (C.22)—$k_{f,i}$ is taken to be the high-pressure reaction coefficient ($k_{\infty,i}$) for falloff reactions and the low-pressure reaction coefficent ($k_{0,i}$) for chemically-activated reactions [173]. The ratio of the coefficients $k_{0,i}$ and

$k_{\infty,i}$, combined with the third-body concentration, define a reduced pressure $P_{r,i}$ given by

$$P_{r,i} = \frac{[X]_i k_{0,i}}{k_{\infty,i}} \tag{C.38}$$

where $[X]_i$ is the appropriate third-body concentration as described in Appendix C.2.6.

The falloff blending factor $F_i$ used in Eq. (C.24) is determined based on one of three representations: the Lindemann [193], Troe [194], and SRI [195] falloff approaches

$$F_i = \begin{cases} 1 & \text{for Lindemann,} \\ F_{\text{cent}}^{\left(1+(A_{\text{Troe}}/B_{\text{Troe}})^2\right)^{-1}} & \text{for Troe, or} \\ dT^e \left( a \cdot \exp\left(-\frac{b}{T}\right) + \exp\left(-\frac{T}{c}\right)\right)^X & \text{for SRI.} \end{cases} \tag{C.39}$$

The Troe representation is described by the variables:

$$F_{cent} = a \exp\left(-\frac{T}{T^*}\right) + (-a+1)\exp\left(-\frac{T}{T^{***}}\right) + \exp\left(-\frac{T^{**}}{T}\right) , \tag{C.40}$$

$$A_{Troe} = -0.67 \log_{10}\left(F_{cent}\right) + \log_{10}\left(P_{r,i}\right) - 0.4 \text{ , and} \tag{C.41}$$

$$B_{Troe} = -1.1762 \log_{10}\left(F_{cent}\right) - 0.14 \log_{10}\left(P_{r,i}\right) + 0.806 , \tag{C.42}$$

where $a$, $T^{***}$, $T^*$, and $T^{**}$ are specified parameters. The final parameter $T^{**}$ is optional, and, if it is not used, the final term of $F_{\text{cent}}$ is omitted.

The exponent used in the SRI representation is given by:

$$X = \frac{1}{\log_{10}^2\left(P_{r,i}\right) + 1} . \tag{C.43}$$

The parameters $a$, $b$, and $c$ in the SRI falloff blending factor are required while $d$ and $e$ are optional; if not specified, $d = 1$ and $e = 0$.

### C.2.8 Pressure-dependent reactions

In addition to the falloff approach given previously, two additional formulations can be used to describe the pressure dependence of reactions that do not follow the modification factor $c_i$ approach. The first involves logarithmic interpolation between Arrhenius rates at two pressures [173, 196] (often termed P-Log reactions), each evaluated using Eq. (C.25):

$$k_1(T) = A_1 T^{\beta_1} \exp\left(-\frac{T_{a,1}}{T}\right) \text{ at } P_1 \text{ and,} \tag{C.44}$$

$$k_2(T) = A_2 T^{\beta_2} \exp\left(-\frac{T_{a,2}}{T}\right) \text{ at } P_2 , \tag{C.45}$$

where the Arrhenius coefficients are given for each pressure $P_1$ and $P_2$. Then, the reaction rate coefficient at a particular pressure $P$ between $P_1$ and $P_2$ can be determined through logarithmic interpolation:

$$\log\left(k_{f,i}\right) = \frac{\left(-\log\left(k_1\right) + \log\left(k_2\right)\right)\left(-\log\left(P_1\right) + \log\left(P\right)\right)}{-\log\left(P_1\right) + \log\left(P_2\right)} + \log\left(k_1\right) \tag{C.46}$$

In addition, the pressure dependence of a reaction can be expressed through a bivariate Chebyshev polynomial fit [173, 196–199]:

$$\log_{10} k_f(T,P) = \sum_{i=1}^{N_T}\sum_{j=1}^{N_p} \eta_{ij}\phi_i(\tilde{T})\phi_j\left(\tilde{P}\right) \ , \tag{C.47}$$

where $\eta_{ij}$ is the coefficient corresponding to the grid points $i$ and $j$, $N_T$ and $N_p$ are the numbers of grid points for temperature and pressure, respectively, and $\phi_n$ is the Chebyshev polynomial of the first kind of degree $n-1$ typically expressed as

$$\phi_n(x) = \mathcal{T}_{n-1}(x) = \cos\left(\left(n-1\right)\arccos(x)\right) \quad \text{for } |x| \leq 1 \ . \tag{C.48}$$

The reduced temperature $\tilde{T}$ and pressure $\tilde{p}$ are given by

$$\tilde{T} \equiv \frac{2T^{-1} - T_{\min}^{-1} - T_{\max}^{-1}}{T_{\max}^{-1} - T_{\min}^{-1}} \quad \text{and,} \tag{C.49}$$

$$\tilde{P} \equiv \frac{2\log_{10} P - \log_{10} P_{\min} - \log_{10} P_{\max}}{\log_{10} P_{\max} - \log_{10} P_{\min}} \ , \tag{C.50}$$

where $T_{\min} \leq T \leq T_{\max}$ and $P_{\min} \leq P \leq P_{\max}$ describe the ranges of validity for temperature and pressure.

## C.3 Jacobian derivation

Let $\mathcal{J}$ denote the Jacobian matrix corresponding to the set of ODEs defined in Eq. (C.3). $\mathcal{J}$ is filled with the partial derivatives $\partial f/\partial\Phi$, such that:

$$\mathcal{J}_{i,j} = \frac{\partial f_i}{\partial \Phi_j}, \qquad i,j = 1 \ldots N_{\text{sp}} + 1 \ . \tag{C.51}$$

where $i$ and $j$ correspond to the row and column, respectively, of the entry in the Jacobian matrix. The Jacobian entries resulting from Eq. (C.51) are derived in Appendices C.3.1 to C.3.3, while various subcomponents of the Jacobian are derived in Appendices C.3.5 to C.3.6. The final form of the Jacobian can be found in Appendix C.4, which provides a useful summary for this

document.

We note that some parts of the Jacobian derivation—in particular derivatives of the energy equation—are quite complicated. The aim of this document is to provide the reader with an overview of the derivation process; some intermediate steps are left out. For a complete derivation, the reader is directed to the output of the derivations.py script, from which this document was compiled.

### C.3.1 Temperature source term derivatives

#### Temperature derivative

First, the temperature source term's derivative with respect to the temperature:

$$\mathcal{J}_{1,1} = \frac{\partial}{\partial T}\frac{\mathrm{d}T}{\mathrm{d}t} \,,$$

or more simply:

$$\mathcal{J}_{1,1} = \frac{\partial \dot{T}}{\partial T} \,, \tag{C.52}$$

will be derived. As the derivation process for this Jacobian entry is very similar between CONP and CONV energy equations, we will focus on the Jacobian entry for the CONP temperature source term and give the result for CONV at the end of this section.

The derivative of a species concentration with respect to temperature for both CONP and CONV is:

$$\frac{\partial [C]_k}{\partial T} = \begin{cases} \dfrac{\partial}{\partial T}\left(\dfrac{n_k}{V}\right) = 0 & \text{for } k \neq N_{\mathrm{sp}} \text{ and,} \\ \dfrac{\partial}{\partial T}\left([C] - \displaystyle\sum_{k=1}^{N_{\mathrm{sp}}-1}\frac{n_k}{V}\right) = -\dfrac{[C]}{T} & \text{for } k = N_{\mathrm{sp}} \,, \end{cases} \tag{C.53}$$

as the pressure and volume are either constants (for CONP and CONV respectively) or a state variable that is a function of time only.

Next, the derivative of full form of the temperature source term Eq. (C.14) with respect to temperature will be considered. To make the result more managable, the denominator of Eq. (C.14) is first collapsed back into the form of Eq. (C.5), yielding for CONP:

$$\frac{\partial \dot{T}}{\partial T} = -\frac{\left([C]\left(\frac{C_{p,N_{\mathrm{sp}}}}{T} - \frac{\mathrm{d}C_{p,N_{\mathrm{sp}}}}{\mathrm{d}T}\right) - \sum_{k=1}^{-1+N_{\mathrm{sp}}}\left(-\frac{\mathrm{d}C_{p,N_{\mathrm{sp}}}}{\mathrm{d}T} + \frac{\mathrm{d}C_{p,k}}{\mathrm{d}T}\right)[C]_k\right)}{\left(\sum_{k=1}^{N_{\mathrm{sp}}}[C]_k C_{p,k}\right)^2} \times$$

$$\sum_{k=1}^{-1+N_{\mathrm{sp}}}\left(H_k-\frac{W_k H_{N_{\mathrm{sp}}}}{W_{N_{\mathrm{sp}}}}\right)\dot{\omega}_k-$$
$$\frac{\sum_{k=1}^{-1+N_{\mathrm{sp}}}\left(\left(H_k-\frac{W_k H_{N_{\mathrm{sp}}}}{W_{N_{\mathrm{sp}}}}\right)\frac{\partial\dot{\omega}_k}{\partial T}+\left(\frac{\mathrm{d}H_k}{\mathrm{d}T}-\frac{W_k}{W_{N_{\mathrm{sp}}}}\frac{\mathrm{d}H_{N_{\mathrm{sp}}}}{\mathrm{d}T}\right)\dot{\omega}_k\right)}{\sum_{k=1}^{N_{\mathrm{sp}}}[C]_k C_{p,k}}\,. \tag{C.54}$$

Next, Eq. (C.5) may be factored out, yielding:

$$\frac{\partial\dot{T}}{\partial T}=\frac{1}{\sum_{k=1}^{N_{\mathrm{sp}}}[C]_k C_{p,k}}\left[\frac{\mathrm{d}T}{\mathrm{d}t}\times\right.$$
$$\left([C]\left(\frac{C_{p,N_{\mathrm{sp}}}}{T}-\frac{\mathrm{d}C_{p,N_{\mathrm{sp}}}}{\mathrm{d}T}\right)-\sum_{k=1}^{-1+N_{\mathrm{sp}}}\left(-\frac{\mathrm{d}C_{p,N_{\mathrm{sp}}}}{\mathrm{d}T}+\frac{\mathrm{d}C_{p,k}}{\mathrm{d}T}\right)[C]_k\right)-$$
$$\left.\sum_{k=1}^{-1+N_{\mathrm{sp}}}\left(\left(H_k-\frac{W_k H_{N_{\mathrm{sp}}}}{W_{N_{\mathrm{sp}}}}\right)\frac{\partial\dot{\omega}_k}{\partial T}+\left(\frac{\mathrm{d}H_k}{\mathrm{d}T}-\frac{W_k}{W_{N_{\mathrm{sp}}}}\frac{\mathrm{d}H_{N_{\mathrm{sp}}}}{\mathrm{d}T}\right)\dot{\omega}_k\right)\right]. \tag{C.55}$$

Now, the identity:

$$[C]=\sum_{k=1}^{N_{\mathrm{sp}}}[C]_k \tag{C.56}$$

is invoked to simplify the specific heat terms:

$$\frac{\partial\dot{T}}{\partial T}=\frac{1}{\sum_{k=1}^{N_{\mathrm{sp}}}[C]_k C_{p,k}}\left[\frac{\mathrm{d}T}{\mathrm{d}t}\sum_{k=1}^{N_{\mathrm{sp}}}\left(-\frac{\mathrm{d}C_{p,k}}{\mathrm{d}T}+\frac{C_{p,N_{\mathrm{sp}}}}{T}\right)[C]_k+\right.$$
$$\sum_{k=1}^{-1+N_{\mathrm{sp}}}\left(\left(-H_k+\frac{W_k H_{N_{\mathrm{sp}}}}{W_{N_{\mathrm{sp}}}}\right)\frac{\partial\dot{\omega}_k}{\partial T}+\right.$$
$$\left.\left.\left(-\frac{\mathrm{d}H_k}{\mathrm{d}T}+\frac{W_k}{W_{N_{\mathrm{sp}}}}\frac{\mathrm{d}H_{N_{\mathrm{sp}}}}{\mathrm{d}T}\right)\dot{\omega}_k\right)\right]. \tag{C.57}$$

Finally, the derivative of species enthalpy with respect to temperature:

$$C_{p,k}=\frac{\mathrm{d}H_k}{\mathrm{d}T}\,, \tag{C.58}$$

is substituted in, giving:

$$\frac{\partial\dot{T}}{\partial T}=\frac{1}{\sum_{k=1}^{N_{\mathrm{sp}}}[C]_k C_{p,k}}\left[\frac{\mathrm{d}T}{\mathrm{d}t}\sum_{k=1}^{N_{\mathrm{sp}}}\left(-\frac{\mathrm{d}C_{p,k}}{\mathrm{d}T}+\frac{C_{p,N_{\mathrm{sp}}}}{T}\right)[C]_k+\right.$$
$$\sum_{k=1}^{-1+N_{\mathrm{sp}}}\left(\left(-H_k+\frac{W_k H_{N_{\mathrm{sp}}}}{W_{N_{\mathrm{sp}}}}\right)\frac{\partial\dot{\omega}_k}{\partial T}+\right.$$

$$\left(-C_{p,k} + \frac{W_k C_{p,k}}{W_{N_{\text{sp}}}}\right)\dot{\omega}_k\Bigg)\Bigg] \qquad \text{for CONP,} \tag{C.59a}$$

and similarly:

$$\begin{aligned}\frac{\partial \dot{T}}{\partial T} = \frac{1}{\sum_{k=1}^{N_{\text{sp}}}[C]_k C_{v,k}}\Bigg[&\frac{\mathrm{d}T}{\mathrm{d}t}\sum_{k=1}^{N_{\text{sp}}}\left(-\frac{\mathrm{d}C_{v,k}}{\mathrm{d}T} + \frac{C_{v,N_{\text{sp}}}}{T}\right)[C]_k + \\ &\sum_{k=1}^{-1+N_{\text{sp}}}\Bigg(\left(-U_k + \frac{W_k U_{N_{\text{sp}}}}{W_{N_{\text{sp}}}}\right)\frac{\partial \dot{\omega}_k}{\partial T} + \\ &\left(-C_{v,k} + \frac{W_k C_{v,N_{\text{sp}}}}{W_{N_{\text{sp}}}}\right)\dot{\omega}_k\Bigg)\Bigg] \qquad \text{for CONV.}\end{aligned} \tag{C.59b}$$

The species production rate derivative with respect to temperature will be defined in Appendix C.3.4.

**State parameter derivatives**

Next, the derivative of the energy equation with respect to the thermodynamic state parameter will be considered:

$$\mathcal{J}_{1,2} = \frac{\partial \dot{T}}{\partial V} \qquad \text{for CONP,} \tag{C.60a}$$

$$\mathcal{J}_{1,2} = \frac{\partial \dot{T}}{\partial P} \qquad \text{for CONV.} \tag{C.60b}$$

First, the derivative of the concentration of a species $k$ with respect to the state parameter is:

$$\frac{\partial [C]_k}{\partial V} = \begin{cases} -\dfrac{[C]_k}{V} & \text{with } k \neq N_{\text{sp}} \\ \dfrac{1}{V}\displaystyle\sum_{k=1}^{N_{\text{sp}}-1}[C]_k & \text{with } k = N_{\text{sp}} \end{cases} \qquad \text{for CONP,} \tag{C.61a}$$

$$\frac{\partial [C]_k}{\partial P} = \begin{cases} 0 & \text{with } k \neq N_{\text{sp}} \\ \dfrac{1}{T\mathcal{R}} & \text{with } k = N_{\text{sp}} \end{cases} \qquad \text{for CONV,} \tag{C.61b}$$

while the specific heat and enthalpy/internal-energy are independent of the state parameter for both CONP and CONV. Following the same procedure as in Appendix C.3.1, the result of Eq. (C.60) is simplified by collapsing the specific-heat/concentration sum, and substituting in

the temperature source term, giving:

$$\frac{\partial \dot{T}}{\partial V} = \frac{1}{\sum_{k=1}^{N_{\text{sp}}} [C]_k C_{p,k}} \Bigg[ - \sum_{k=1}^{-1+N_{\text{sp}}} \left( H_k - \frac{W_k H_{N_{\text{sp}}}}{W_{N_{\text{sp}}}} \right) \frac{\partial \dot{\omega}_k}{\partial V} + \frac{1}{V} \frac{\mathrm{d}T}{\mathrm{d}t} \sum_{k=1}^{-1+N_{\text{sp}}} \left( -C_{p,N_{\text{sp}}} + C_{p,k} \right) [C]_k \Bigg] \qquad \text{for CONP,} \tag{C.62a}$$

$$\frac{\partial \dot{T}}{\partial P} = \frac{1}{\sum_{k=1}^{N_{\text{sp}}} [C]_k C_{v,k}} \Bigg[ - \sum_{k=1}^{-1+N_{\text{sp}}} \left( U_k - \frac{W_k U_{N_{\text{sp}}}}{W_{N_{\text{sp}}}} \right) \frac{\partial \dot{\omega}_k}{\partial P} - \frac{C_{v,N_{\text{sp}}}}{T\mathcal{R}} \frac{\mathrm{d}T}{\mathrm{d}t} \Bigg] \qquad \text{for CONV.} \tag{C.62b}$$

The derivative of the species production rate with respect to the state parameter will be computed in Appendix C.3.4.

### Molar derivative

Finally, the derivative of the temperature source term with respect to the moles of species $j$ is:

$$\mathcal{J}_{1,j+2} = \frac{\partial \dot{T}}{\partial n_j} \tag{C.63}$$

First, the molar derivative of a species concentration is computed as:

$$\frac{\partial [C]_k}{\partial n_j} = \begin{cases} \dfrac{\delta_{jk}}{V} & \text{for } k \neq N_{\text{sp}} \\ -\dfrac{1}{V} & \text{for } k = N_{\text{sp}} \end{cases} \tag{C.64}$$

Or put succinctly:

$$\frac{\partial [C]_k}{\partial n_j} = \frac{\delta_{jk} - \delta_{N_{\text{sp}} k}}{V} \tag{C.65}$$

The derivative of the energy equation with respect to the moles of species $j$—after collapsing the specific-heat/concentration sum, as in Appendix C.3.1—is given by:

$$\frac{\partial \dot{T}}{\partial n_j} = \frac{\left( \sum_{k=1}^{-1+N_{\text{sp}}} \left( H_k - \frac{W_k H_{N_{\text{sp}}}}{W_{N_{\text{sp}}}} \right) \dot{\omega}_k \right) \sum_{k=1}^{-1+N_{\text{sp}}} -\frac{\delta_{jk}}{V} \left( C_{p,N_{\text{sp}}} - C_{p,k} \right)}{\left( \sum_{k=1}^{N_{\text{sp}}} [C]_k C_{p,k} \right)^2} - \frac{1}{\sum_{k=1}^{N_{\text{sp}}} [C]_k C_{p,k}} \sum_{k=1}^{-1+N_{\text{sp}}} \left( H_k - \frac{W_k H_{N_{\text{sp}}}}{W_{N_{\text{sp}}}} \right) \frac{\partial \dot{\omega}_k}{\partial n_j} . \tag{C.66}$$

Next, the Kronecker delta summation is simplified, and the compact form of the energy equation

(Eq. (C.5)) factored out yielding:

$$\frac{\partial \dot{T}}{\partial n_j} = \frac{1}{\sum_{k=1}^{N_{\text{sp}}} [C]_k C_{p,k}} \left[ - \sum_{k=1}^{-1+N_{\text{sp}}} \left( H_k - \frac{W_k H_{N_{\text{sp}}}}{W_{N_{\text{sp}}}} \right) \frac{\partial \dot{\omega}_k}{\partial n_j} - \frac{1}{V} \frac{\mathrm{d}T}{\mathrm{d}t} \left( -C_{p,N_{\text{sp}}} + C_{p,j} \right) \right] \qquad \text{for CONP,} \tag{C.67a}$$

and:

$$\frac{\partial \dot{T}}{\partial n_j} = \frac{1}{\sum_{k=1}^{N_{\text{sp}}} [C]_k C_{v,k}} \left[ - \sum_{k=1}^{-1+N_{\text{sp}}} \left( U_k - \frac{W_k U_{N_{\text{sp}}}}{W_{N_{\text{sp}}}} \right) \frac{\partial \dot{\omega}_k}{\partial n_j} - \frac{1}{V} \frac{\mathrm{d}T}{\mathrm{d}t} \left( -C_{v,N_{\text{sp}}} + C_{v,j} \right) \right] \qquad \text{for CONV.} \tag{C.67b}$$

Again, the derivative of the species production rate with respect to moles will be explored in Appendix C.3.4.

### C.3.2 State parameter source term derivatives

#### Temperature derivative

The derivatives of the thermodynamic state parameter source terms with respect to temperature are given by differentiating Eq. (C.11):

$$\begin{aligned} \mathcal{J}_{2,1} &= \frac{\partial}{\partial T} \frac{\mathrm{d}V}{\mathrm{d}t} = \frac{\partial \dot{V}}{\partial T} \\ &= \frac{V}{[C]} \sum_{k=1}^{-1+N_{\text{sp}}} \left( 1 - \frac{W_k}{W_{N_{\text{sp}}}} \right) \left( \frac{\partial \dot{\omega}_k}{\partial T} + \frac{\dot{\omega}_k}{T} \right) \\ &\quad + \frac{V}{T} \left( \frac{\mathrm{d}\dot{T}}{\mathrm{d}T} - \frac{1}{T} \frac{\mathrm{d}T}{\mathrm{d}t} \right) \qquad \text{for CONP,} \end{aligned} \tag{C.68a}$$

and:

$$\begin{aligned} \mathcal{J}_{2,1} &= \frac{\partial}{\partial T} \frac{\mathrm{d}V}{\mathrm{d}t} = \frac{\partial \dot{P}}{\partial T} \\ &= \mathcal{R} \sum_{k=1}^{-1+N_{\text{sp}}} \left( 1 - \frac{W_k}{W_{N_{\text{sp}}}} \right) \left( T \frac{\partial \dot{\omega}_k}{\partial T} + \dot{\omega}_k \right) \\ &\quad + \frac{P}{T} \left( \frac{\mathrm{d}\dot{T}}{\mathrm{d}T} - \frac{1}{T} \frac{\mathrm{d}T}{\mathrm{d}t} \right) \qquad \text{for CONV.} \end{aligned} \tag{C.68b}$$

**State parameter derivative**

Similarly, the derivatives of the thermodynamic state parameter source terms with respect to themselves are:

$$\begin{aligned}\mathcal{J}_{2,2} &= \frac{\partial}{\partial V}\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{\partial \dot{V}}{\partial V} \\ &= \frac{1}{[C]}\sum_{k=1}^{-1+N_{\mathrm{sp}}}\left(1-\frac{W_k}{W_{N_{\mathrm{sp}}}}\right)\left(V\frac{\partial \dot{\omega}_k}{\partial V}+\dot{\omega}_k\right) \\ &\quad + \frac{1}{T}\left(V\frac{\mathrm{d}\dot{T}}{\mathrm{d}V}+\dot{T}\right) \qquad \text{for CONP,} \end{aligned} \tag{C.69a}$$

and:

$$\begin{aligned}\mathcal{J}_{2,2} &= \frac{\partial}{\partial P}\frac{\mathrm{d}P}{\mathrm{d}t} = \frac{\partial \dot{P}}{\partial P} \\ &= T\mathcal{R}\sum_{k=1}^{-1+N_{\mathrm{sp}}}\left(1-\frac{W_k}{W_{N_{\mathrm{sp}}}}\right)\frac{\partial \dot{\omega}_k}{\partial P} \\ &\quad + \frac{1}{T}\left(P\frac{\mathrm{d}\dot{T}}{\mathrm{d}P}+\dot{T}\right) \qquad \text{for CONV.} \end{aligned} \tag{C.69b}$$

**Molar derivative**

Finally, the derivatives of the thermodynamic state parameter source terms with respect to the moles of species $j$ are:

$$\begin{aligned}\mathcal{J}_{2,j+2} &= \frac{\partial}{\partial n_j}\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{\partial \dot{V}}{\partial n_j} \\ &= V\left(\frac{1}{[C]}\sum_{k=1}^{-1+N_{\mathrm{sp}}}\left(1-\frac{W_k}{W_{N_{\mathrm{sp}}}}\right)\frac{\partial \dot{\omega}_k}{\partial n_j}+\frac{1}{T}\frac{\mathrm{d}\dot{T}}{\mathrm{d}n_j}\right) \qquad \text{for CONP,} \end{aligned} \tag{C.70a}$$

and:

$$\begin{aligned}\mathcal{J}_{2,j+2} &= \frac{\partial}{\partial n_j}\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{\partial \dot{P}}{\partial n_j} \\ &= \frac{P}{T}\frac{\mathrm{d}\dot{T}}{\mathrm{d}n_j}+T\mathcal{R}\sum_{k=1}^{-1+N_{\mathrm{sp}}}\left(1-\frac{W_k}{W_{N_{\mathrm{sp}}}}\right)\frac{\partial \dot{\omega}_k}{\partial n_j} \qquad \text{for CONV.} \end{aligned} \tag{C.70b}$$

### C.3.3 Molar source term derivatives

**Temperature derivative**

The derivative of the net molar rate of production of a species $k$ with respect to temperature is:

$$\mathcal{J}_{k+2,1} = \frac{\partial}{\partial T}\frac{\mathrm{d}n_k}{\mathrm{d}t} = \frac{\partial \dot{n}_k}{\partial T}\ ,$$

or:

$$\mathcal{J}_{k+2,1} = V\frac{\partial\dot{\omega}_k}{\partial T} \ . \tag{C.71}$$

**State parameter derivatives**

Similarly, the derivative of the net molar rate of production of a species $k$ with respect to the state parameter is:

$$\begin{aligned}\mathcal{J}_{k+2,2} &= \frac{\partial}{\partial V}\frac{\mathrm{d}n_k}{\mathrm{d}t} = \frac{\partial\dot{n}_k}{\partial V} \\ &= V\frac{\partial\dot{\omega}_k}{\partial V} + \dot{\omega}_k \qquad \text{for CONP,}\end{aligned} \tag{C.72a}$$

and:

$$\begin{aligned}\mathcal{J}_{k+2,2} &= \frac{\partial}{\partial P}\frac{\mathrm{d}n_k}{\mathrm{d}t} = \frac{\partial\dot{n}_k}{\partial P} \\ &= V\frac{\partial\dot{\omega}_k}{\partial P} \qquad \text{for CONV.}\end{aligned} \tag{C.72b}$$

**Molar derivative**

Finally, the derivative of the net molar rate of production of a species $k$ with respect to the moles of species $j$ is:

$$\mathcal{J}_{k+2,j+2} = \frac{\partial}{\partial n_j}\frac{\mathrm{d}n_k}{\mathrm{d}t} = \frac{\partial\dot{n}_k}{\partial n_j} \ ,$$

or:

$$\mathcal{J}_{k+2,j+2} = V\frac{\partial\dot{\omega}_k}{\partial n_j} \ . \tag{C.73}$$

### C.3.4 Species production rate derivatives

The derivatives of the net species production rate equation Eq. (C.18), with respect to temperature are:

$$\frac{\partial\dot{\omega}_k}{\partial T} = \sum_{i=1}^{N_{\text{reac}}}\left(\nu_{k,i}R_i\frac{\partial c_i}{\partial T} + \nu_{k,i}\frac{\partial R_i}{\partial T}c_i\right) \ , \tag{C.74}$$

the state parameter:

$$\frac{\partial\dot{\omega}_k}{\partial V} = \sum_{i=1}^{N_{\text{reac}}}\left(\nu_{k,i}R_i\frac{\partial c_i}{\partial V} + \nu_{k,i}\frac{\partial R_i}{\partial V}c_i\right) \qquad \text{for CONP,} \tag{C.75a}$$

$$\frac{\partial \dot{\omega}_k}{\partial P} = \sum_{i=1}^{N_{\text{reac}}} \left( \nu_{k,i} R_i \frac{\partial c_i}{\partial P} + \nu_{k,i} \frac{\partial R_i}{\partial P} c_i \right) \qquad \text{for CONV,} \tag{C.75b}$$

and other species:

$$\frac{\partial \dot{\omega}_k}{\partial n_j} = \sum_{i=1}^{N_{\text{reac}}} \left( \nu_{k,i} R_i \frac{\partial c_i}{\partial n_j} + \nu_{k,i} \frac{\partial R_i}{\partial n_j} c_i \right) \quad . \tag{C.76}$$

### C.3.5 Rate of progress derivatives

#### Temperature derivative

Next, the rate of progress derivatives are evaluated, first with respect to temperature:

$$\frac{\partial R_i}{\partial T} = \frac{\partial}{\partial T} \left( k_{f,i} \prod_{k=1}^{N_{\text{sp}}} [C]_k^{\nu'_{k,i}} \right) \quad . \tag{C.77}$$

The derivative of the forward reaction rate coefficient is:

$$\frac{\mathrm{d}k_{f,i}}{\mathrm{d}T} = \frac{k_{f,i}}{T} \left( \beta_i + \frac{E_{ai}}{T\mathcal{R}} \right) \quad . \tag{C.78}$$

While evaluating derivatives for the Jacobian, it is important to expand terms—e.g., concentration, production rate, etc.—related to the last species to obtain the correct derivative, in this case using Eq. (C.12):

$$R_f = \left( \left( - \sum_{k=1}^{N_{\text{sp}}-1} [C]_k + \frac{P}{T\mathcal{R}} \right)^{\nu'_{N_{\text{sp}},i}} \right) k_{f,i} \prod_{k=1}^{N_{\text{sp}}-1} [C]_k^{\nu'_{k,i}} \quad . \tag{C.79}$$

Hence, the full derivative of the forward rate of progress is:

$$\frac{\partial R_{f\,i}}{\partial T} = \frac{\mathrm{d}k_{f,i}}{\mathrm{d}T} \prod_{k=1}^{N_{\text{sp}}} [C]_k^{\nu'_{k,i}} - \frac{[C]\nu'_{N_{\text{sp}},i}}{T} [C]_{N_{\text{sp}}}^{\nu'_{N_{\text{sp}},i}-1} k_{f,i} \prod_{k=1}^{N_{\text{sp}}-1} [C]_k^{\nu'_{k,i}} \quad . \tag{C.80}$$

By defining a temporary variable $S'_l$ (for species $l = \{j, N_{\text{sp}}\}$):

$$S'_l = \nu'_{l,i} [C]_l^{\nu'_{l,i}-1} \prod_{\substack{1 \le l \le l-1 \\ l+1 \le l \le N_{\text{sp}}}} [C]_l^{\nu'_{l,i}} \quad , \tag{C.81}$$

and using Eq. (C.78), Eq. (C.80) can be simplified to:

$$\frac{\partial R_{f\,i}}{\partial T} = -\frac{[C] S'_{N_{\text{sp}}}}{T} k_{f,i} + \frac{R_{f,i}}{T} \left( \beta_i + \frac{E_{ai}}{T\mathcal{R}} \right) \quad . \tag{C.82}$$

Starting from Eq. (C.23) and applying the same expansion of the $N_{\text{sp}}$-th species' concentration as previously, the temperature derivative of the reverse rate of progress is:

$$\frac{\partial R_{r,i}}{\partial T} = \frac{\mathrm{d}k_{r,i}}{\mathrm{d}T}\prod_{k=1}^{N_{\text{sp}}}[C]_k^{\nu''_{k,i}} - \frac{[C]\nu''_{N_{\text{sp}},i}}{T}[C]_{N_{\text{sp}}}^{\nu''_{N_{\text{sp}},i}-1}k_{f,i}\prod_{k=1}^{N_{\text{sp}}-1}[C]_k^{\nu''_{k,i}} \,. \tag{C.83}$$

Next, Eq. (C.27) is considered to obtain the temperature derivative of the non-explicit reversible reaction rate coefficient:

$$\begin{aligned}\frac{\mathrm{d}k_{r,i}}{\mathrm{d}T} &= \left(-\frac{1}{K_{c,i}}\frac{\mathrm{d}K_{c,i}}{\mathrm{d}T}\frac{k_{f\,i}}{K_{ci}} + \frac{1}{K_{c,i}}\frac{\mathrm{d}k_{f,i}}{\mathrm{d}T}\right) \,, \\ &= \left(-\frac{1}{K_{c,i}}\frac{\mathrm{d}K_{c,i}}{\mathrm{d}T} + \frac{1}{T}\left(\beta_i + \frac{E_{ai}}{T\mathcal{R}}\right)\right)k_{r,i} \,.\end{aligned} \tag{C.84}$$

The temperature derivative of the equilibrium constant is:

$$\frac{\mathrm{d}K_{c,i}}{\mathrm{d}T} = K_{c,i}\sum_{k=1}^{N_{\text{sp}}}\nu_{k,i}\frac{\mathrm{d}B_k}{\mathrm{d}T} \,, \tag{C.85}$$

with:

$$\frac{\mathrm{d}B_k}{\mathrm{d}T} = T\left(T\left(\frac{Ta_{k,4}}{5} + \frac{a_{k,3}}{4}\right) + \frac{a_{k,2}}{3}\right) + \frac{a_{k,1}}{2} + \frac{1}{T}\left(a_{k,0} - 1 + \frac{a_{k,5}}{T}\right) \,, \tag{C.86}$$

giving:

$$\frac{\mathrm{d}k_{r,i}}{\mathrm{d}T} = \left(-\sum_{k=1}^{N_{\text{sp}}}\nu_{k,i}\frac{\mathrm{d}B_k}{\mathrm{d}T} + \frac{1}{T}\left(\beta_i + \frac{E_{ai}}{T\mathcal{R}}\right)\right)k_{r,i} \,. \tag{C.87}$$

Using a temporary value $S''_{N_{\text{sp}}}$—defined analagously to Eq. (C.81) for the reverse reaction rate—the full deriviative of the reverse rate of progress with respect to temperature is obtained:

$$\frac{\partial R_{r,i}}{\partial T} = \left(-\sum_{k=1}^{N_{\text{sp}}}\nu_{k,i}\frac{\mathrm{d}B_k}{\mathrm{d}T} + \frac{1}{T}\left(\beta_i + \frac{E_{ai}}{T\mathcal{R}}\right)\right)R_{r,i} - \frac{[C]S''_{N_{\text{sp}}}}{T}k_{r,i} \,. \tag{C.88}$$

Finally, combining Eqs. (C.82) and (C.88) gives the total temperature derivative of the net rate of progress:

$$\begin{aligned}\frac{\partial R_i}{\partial T} =& \frac{R_{f,i}}{T}\left(\beta_i + \frac{E_{ai}}{T\mathcal{R}}\right) - \left(-\sum_{k=1}^{N_{\text{sp}}}\nu_{k,i}\frac{\mathrm{d}B_k}{\mathrm{d}T} + \frac{1}{T}\left(\beta_i + \frac{E_{ai}}{T\mathcal{R}}\right)\right)R_{r,i} \\ &+ \frac{[C]S''_{N_{\text{sp}}}}{T}k_{r,i} - \frac{[C]S'_{N_{\text{sp}}}}{T}k_{f,i} \,.\end{aligned} \tag{C.89}$$

**State parameter derivatives**

Following the outline of Appendix C.3.5—while utilising Eq. (C.61) for the species concentration derivatives—the forward rate of progress derivatives are:

$$\frac{\partial R_{f,i}}{\partial V} = \frac{[C]S'_{N_{\mathrm{sp}}}}{V}k_{f,i} - \frac{R_{f,i}}{V}\sum_{k=1}^{N_{\mathrm{sp}}}\nu'_{k,i} \qquad \text{for CONP,} \tag{C.90a}$$

$$\frac{\partial R_{f,i}}{\partial P} = \frac{S'_{N_{\mathrm{sp}}}k_{f,i}}{T\mathcal{R}} \qquad \text{for CONV.} \tag{C.90b}$$

Similarly, the reverse rate of progress derivatives are:

$$\frac{\partial R_{r,i}}{\partial V} = \frac{[C]S''_{N_{\mathrm{sp}}}}{V}k_{r,i} - \frac{R_{r,i}}{V}\sum_{k=1}^{N_{\mathrm{sp}}}\nu''_{k,i} \qquad \text{for CONP,} \tag{C.91a}$$

and:

$$\frac{\partial R_{r,i}}{\partial P} = \frac{S''_{N_{\mathrm{sp}}}k_{r,i}}{T\mathcal{R}} \qquad \text{for CONV.} \tag{C.91b}$$

Combining Eqs. (C.90) and (C.91) gives:

$$\frac{\partial R_i}{\partial V} = \frac{1}{V}\left([C]\left(S'_{N_{\mathrm{sp}}}k_{f,i} - S''_{N_{\mathrm{sp}}}k_{r,i}\right) + \left(R_{r,i}\sum_{k=1}^{N_{\mathrm{sp}}}\nu''_{k,i} - R_{f,i}\sum_{k=1}^{N_{\mathrm{sp}}}\nu'_{k,i}\right)\right) \qquad \text{for CONP,} \tag{C.92a}$$

and:

$$\frac{\partial R_i}{\partial P} = \frac{1}{T\mathcal{R}}\left(S'_{N_{\mathrm{sp}}}k_{f,i} - S''_{N_{\mathrm{sp}}}k_{r,i}\right) \qquad \text{for CONV.} \tag{C.92b}$$

**Molar derivative**

The derivative of the forward rate of progress with respect to the amount of moles of a species $j$ is:

$$\frac{d}{dn_k}R_{f_i} = \left(\frac{\partial}{\partial n_j}\prod_{k=1}^{N_{\mathrm{sp}}}[C]_k^{\nu'_{k,i}}\right)k_{f,i}\ . \tag{C.93}$$

Combining Eqs. (C.65) and (C.93) the molar derivative of the forward rate of progress is:

$$\frac{\partial R_{f,i}}{\partial n_j} = k_{f,i}\sum_{k=1}^{N_{\mathrm{sp}}}\left(-\frac{\delta_{N_{\mathrm{sp}}k}}{V} + \frac{\delta_{jk}}{V}\right)\nu'_{k,i}[C]_k^{\nu'_{k,i}-1}\prod_{\substack{1\le l\le k-1\\ k+1\le l\le N_{\mathrm{sp}}}}[C]_l^{\nu'_{l,i}} \tag{C.94}$$

with the same temporary variables $S'_j$ and $S'_{N_{\text{sp}}}$ as defined previously, this can be simplified to:

$$\frac{\partial R_{f,i}}{\partial n_j} = \frac{k_{f,i}}{V}\left(-S'_{N_{\text{sp}}} + S'_j\right) \ . \tag{C.95}$$

By the same process, the reverse rate of progress derivative is calculated as:

$$\frac{\partial R_{r,i}}{\partial n_j} = \frac{k_{r,i}}{V}\left(-S''_{N_{\text{sp}}} + S''_j\right) \ , \tag{C.96}$$

and from Eqs. (C.95) and (C.96) the net rate of progress molar derivative is found:

$$\frac{\partial R_i}{\partial n_j} = -\frac{k_{r,i}}{V}\left(-S''_{N_{\text{sp}}} + S''_j\right) + \frac{k_{f,i}}{V}\left(-S'_{N_{\text{sp}}} + S'_j\right) \ . \tag{C.97}$$

**Pressure-dependent reactions**

The P-Log and Chebyshev pressure-dependent reactions, described by Eqs. (C.46) and (C.47) require separate treatment from the derivatives of strictly Arrhenius-based reaction rate formulations examined in Appendix C.3.5. Beginning with P-Log reactions, the derivative of the forward rate coefficient with respect to temperature is:

$$\begin{aligned}\frac{\partial k_{f,i}}{\partial T} = \Bigg(&\frac{1}{k_1}\frac{\mathrm{d}k_1}{\mathrm{d}T} + \frac{1}{-\log\left(P_1\right) + \log\left(P_2\right)}\left(-\frac{1}{k_1}\frac{\mathrm{d}k_1}{\mathrm{d}T} + \frac{1}{k_2}\frac{\mathrm{d}k_2}{\mathrm{d}T}\right) \times \Big( \\ &- \log\left(P_1\right) + \log\left(P\right)\Big)\Bigg) k_{f,i} \ ,\end{aligned} \tag{C.98}$$

and the derivatives of $k_1$ and $k_2$ are given by Eq. (C.78) with the corresponding Arrhenius parameters (see Eqs. (C.44) and (C.45)) substituted in.
Simplifying Eq. (C.98) gives:

$$\begin{aligned}\frac{\partial k_{f,i}}{\partial T} = \frac{k_{f,i}}{T}\Bigg(&\frac{\left(-\log\left(P_1\right) + \log\left(P\right)\right)}{-\log\left(P_1\right) + \log\left(P_2\right)} \times \\ &\left(-\beta_1 + \beta_2 - \frac{E_{a_1}}{T\mathcal{R}} + \frac{E_{a_2}}{T\mathcal{R}}\right) + \beta_1 + \frac{E_{a_1}}{T\mathcal{R}}\Bigg) \ .\end{aligned} \tag{C.99}$$

Similar to Appendix C.3.5, Eq. (C.99) may be substituted into Eqs. (C.80) and (C.83) to give:

$$\begin{aligned}\frac{\partial R_i}{\partial T} = &\left(\frac{\left(-\log\left(P_1\right) + \log\left(P\right)\right)\left(-\beta_1 + \beta_2 - \frac{E_{a_1}}{T\mathcal{R}} + \frac{E_{a_2}}{T\mathcal{R}}\right)}{-\log\left(P_1\right) + \log\left(P_2\right)} + \beta_1 + \frac{E_{a_1}}{T\mathcal{R}}\right) \times \\ &\left(\frac{R_{f,i} - R_{r,i}}{T}\right) + R_{r,i}\sum_{k=1}^{N_{\text{sp}}}\nu_{k,i}\frac{\mathrm{d}B_k}{\mathrm{d}T} + \frac{[C]}{T}\left(S''_{N_{\text{sp}}}k_{r,i} - S'_{N_{\text{sp}}}k_{f,i}\right) \ .\end{aligned} \tag{C.100}$$

For CONV problems, the P-Log forward reaction rate coefficient is also a function of the pressure:

$$\frac{\partial k_{f,i}}{\partial P} = \frac{\left(-\log\left(k_1\right) + \log\left(k_2\right)\right) k_{f,i}}{P\left(-\log\left(P_1\right) + \log\left(P_2\right)\right)} \ , \tag{C.101}$$

substituting this into the net rate of progress derivative gives:

$$\begin{aligned} \frac{\partial R_i}{\partial P} = & \frac{1}{T\mathcal{R}}\left(-S''_{N_{\text{sp}}} k_{r,i} + S'_{N_{\text{sp}}} k_{f,i}\right) + \\ & \frac{\left(-\log\left(k_1\right) + \log\left(k_2\right)\right)\left(R_{f,i} - R_{r,i}\right)}{P\left(-\log\left(P_1\right) + \log\left(P_2\right)\right)} \qquad \text{for CONV.} \end{aligned} \tag{C.102}$$

Next, the temperature derivative of Chevbyshev reactions will be considered:

$$\frac{\partial k_{f,i}}{\partial T} = \ln\left(10\right) k_{f,i} \sum_{\substack{1 \le l \le N_P \\ 1 \le j \le N_T}} \frac{\mathrm{d}\tilde{T}}{\mathrm{d}T}\left(j-1\right) \mathcal{T}_{l-1}\left(\tilde{P}\right) U_{j-2}\left(\tilde{T}\right) \eta_{l,j} \ , \tag{C.103}$$

where $U_n$ is the Chebyshev polynomial of the second kind of degree $n$, expressed as:

$$U_n\left(x\right) = \frac{\sin\left(\left(n+1\right)\arccos x\right)}{\sin\left(\arccos x\right)} \tag{C.104}$$

and:

$$\frac{\mathrm{d}\tilde{T}}{\mathrm{d}T} = \frac{-2}{T^2\left(-\frac{1}{T_{\min}} + \frac{1}{T_{\max}}\right)} \ , \tag{C.105}$$

giving:

$$\frac{\partial k_{f,i}}{\partial T} = \ln\left(10\right) k_{f,i} \sum_{\substack{1 \le l \le N_P \\ 1 \le j \le N_T}} -\frac{2\mathcal{T}_{l-1}\left(\tilde{P}\right) U_{j-2}\left(\tilde{T}\right) \eta_{l,j}}{T^2\left(-\frac{1}{T_{\min}} + \frac{1}{T_{\max}}\right)}\left(j-1\right) \ . \tag{C.106}$$

Following the same process for Arrhenius-based reactions (see Appendix C.3.5), the derivative of the net rate of progress with respect to temperature is:

$$\begin{aligned} \frac{\partial R_i}{\partial T} = & \sum_{\substack{1 \le l \le N_P \\ 1 \le j \le N_T}} \left[-\frac{2\ln\left(10\right)\mathcal{T}_{l-1}\left(\tilde{P}\right) U_{j-2}\left(\tilde{T}\right) \eta_{l,j}}{T^2\left(-\frac{1}{T_{\min}} + \frac{1}{T_{\max}}\right)}\left(j-1\right)\right]\left(R_{f,i} - R_{r,i}\right) + \\ & \sum_{k=1}^{N_{\text{sp}}} \nu_{k,i} \frac{\mathrm{d}B_k}{\mathrm{d}T} R_{r,i} + \frac{[C]}{T}\left(S''_{N_{\text{sp}}} k_{r,i} - S'_{N_{\text{sp}}} k_{f,i}\right) \ . \end{aligned} \tag{C.107}$$

For CONV problems, the derivative of the forward rate coefficient with respect to pressure must again be considered:

$$\frac{\partial k_{f,i}}{\partial P} = \ln\left(10\right) k_{f,i} \sum_{\substack{1 \le l \le N_P \\ 1 \le j \le N_T}} \frac{\mathrm{d}\tilde{P}}{\mathrm{d}P}\left(l-1\right) \mathcal{T}_{j-1}\left(\tilde{T}\right) U_{l-2}\left(\tilde{P}\right) \eta_{l,j} \ , \tag{C.108}$$

with:

$$\frac{\mathrm{d}\tilde{P}}{\mathrm{d}P} = \frac{2}{P\left(\log\left(P_{\max}\right) - \log\left(P_{\min}\right)\right)}\ , \tag{C.109}$$

giving:

$$\frac{\partial k_{f,i}}{\partial P} = \ln(10) k_{f,i} \sum_{\substack{1\leq l\leq N_P \\ 1\leq j\leq N_T}} \frac{2\left(l-1\right)\mathcal{T}_{j-1}\left(\tilde{T}\right)U_{l-2}\left(\tilde{P}\right)\eta_{l,j}}{P\left(\log\left(P_{\max}\right) - \log\left(P_{\min}\right)\right)}\ . \tag{C.110}$$

Finally, the net rate of progress derivative for a Chebyshev reaction with respect to pressure is:

$$\begin{aligned}\frac{\partial R_i}{\partial P} = \ln(10)\left(R_{f,i} - R_{r,i}\right) \times & \\ \sum_{\substack{1\leq l\leq N_P \\ 1\leq j\leq N_T}} \frac{2\left(l-1\right)\mathcal{T}_{j-1}\left(\tilde{T}\right)U_{l-2}\left(\tilde{P}\right)\eta_{l,j}}{P\left(\log\left(P_{\max}\right) - \log\left(P_{\min}\right)\right)} + & \\ \frac{1}{T\mathcal{R}}\left(-S''_{N_{\mathrm{sp}}}k_{r,i} + S'_{N_{\mathrm{sp}}}k_{f,i}\right) \qquad \text{for CONV.}\end{aligned} \tag{C.111}$$

### C.3.6 Pressure modification/Falloff function derivatives

#### Elementary reactions

For elementary reactions, the derivative of the pressure-modification term with respect to temperature is:

$$\frac{\partial c_i}{\partial T} = 0 \qquad , \tag{C.112}$$

and with respect to the state parameter:

$$\frac{\partial c_i}{\partial V} = 0 \qquad \text{for CONP,} \tag{C.113a}$$

$$\frac{\partial c_i}{\partial P} = 0 \qquad \text{for CONV,} \tag{C.113b}$$

and finally, with respect to the moles of species $j$:

$$\frac{\partial c_i}{\partial n_j} = 0 \qquad . \tag{C.114}$$

#### Third-body enhanced reactions

For a **mixture** based third-body enhanced reaction Eq. (C.34) is differentiated with respect to temperature to give:

$$\frac{\partial [X]_i}{\partial T} = -\frac{[C]\alpha_{N_{\mathrm{sp}},i}}{T} \qquad , \tag{C.115}$$

and with respect to the state parameter:

$$\frac{\partial [X]_i}{\partial V} = \frac{1}{V}\left([C]\alpha_{N_{\mathrm{sp}},i} - [X]_i\right) \qquad \text{for CONP,} \tag{C.116a}$$

$$\frac{\partial [X]_i}{\partial P} = \frac{\alpha_{N_{\mathrm{sp}},i}}{T\mathcal{R}} \qquad \text{for CONV,} \tag{C.116b}$$

and finally, with respect to the moles of species $j$:

$$\frac{\partial [X]_i}{\partial n_j} = \frac{1}{V}\left(-\alpha_{N_{\mathrm{sp}},i} + \alpha_{j,i}\right) \quad . \tag{C.117}$$

If $\alpha_{j,i} = 1$ for all species $j$ (i.e., a **unity** third-body), Eqs. (C.115) to (C.117) may be simplified to:

$$\frac{\partial c_i}{\partial T} = -\frac{[C]}{T} \quad , \tag{C.118}$$

and for the state parameter:

$$\frac{\partial c_i}{\partial V} = 0 \qquad \text{for CONP,} \tag{C.119a}$$

$$\frac{\partial c_i}{\partial P} = \frac{1}{T\mathcal{R}} \qquad \text{for CONV,} \tag{C.119b}$$

and finally for the moles of species $j$:

$$\frac{\partial c_i}{\partial n_j} = 0 \quad . \tag{C.120}$$

If the third-body is the $m$th **species**, Eqs. (C.115) to (C.117) become:

$$\frac{\partial c_i}{\partial T} = -\frac{\delta_{N_{\mathrm{sp}}m}}{T}[C] \quad , \tag{C.121}$$

and:

$$\frac{\partial c_i}{\partial V} = \frac{\delta_{N_{\mathrm{sp}}m}}{V}\left([C] - [C]_{N_{\mathrm{sp}}}\right) + \frac{[C]_m}{V}\left(\delta_{N_{\mathrm{sp}}m} - 1\right) \qquad \text{for CONP,} \tag{C.122a}$$

$$\frac{\partial c_i}{\partial P} = \frac{\delta_{N_{\mathrm{sp}}m}}{T\mathcal{R}} \qquad \text{for CONV,} \tag{C.122b}$$

and for the molar derivative:

$$\frac{\partial c_i}{\partial n_j} = \frac{1}{V}\left(-\delta_{N_{\mathrm{sp}}m} + \delta_{jm}\right) \quad . \tag{C.123}$$

### Unimolecular/recombination falloff reactions

Derivatives of the third-body pressure modification term for unimolecular/recombination falloff reactions—given by Eq. (C.24)—are quite involved, and will be broken up over the next sections. The derivative of full third-body pressure modification term for falloff reactions will be given in this section, while Appendix C.3.6 will provide the same derivatives for chemically-activated bimolecular reactions. Subsequently, the derivatives of the reduced pressure and falloff blending factors will be explored in Appendix C.3.6 respectively.

The derivatives of the unimolecular/recombination falloff reaction third-body pressure modification term are:

$$\frac{\partial c_i}{\partial T} = \frac{1}{P_{r,i}+1}\left(P_{r,i}\frac{\partial F_i}{\partial T} + \frac{\partial P_{r,i}}{\partial T}\left(F_i - c_i\right)\right) \quad , \tag{C.124}$$

and with respect to the state parameter:

$$\frac{\partial c_i}{\partial V} = \frac{1}{P_{r,i}+1}\left(P_{r,i}\frac{\partial F_i}{\partial V} + \frac{\partial P_{r,i}}{\partial V}\left(F_i - c_i\right)\right) \quad \text{for CONP,} \tag{C.125a}$$

$$\frac{\partial c_i}{\partial P} = \frac{1}{P_{r,i}+1}\left(P_{r,i}\frac{\partial F_i}{\partial P} + \frac{\partial P_{r,i}}{\partial P}\left(F_i - c_i\right)\right) \quad \text{for CONV,} \tag{C.125b}$$

and finally the molar derivative:

$$\frac{\partial c_i}{\partial n_j} = \frac{1}{P_{r,i}+1}\left(P_{r,i}\frac{\partial F_i}{\partial n[j]} + \frac{\partial P_{r,i}}{\partial n_j}\left(F_i - c_i\right)\right) \quad . \tag{C.126}$$

### Chemically-activated bimolecular reactions

Similarly, the chemically-activated bimolecular reactions are as follows:

$$\frac{\partial c_i}{\partial T} = \frac{1}{P_{r,i}+1}\left(\frac{\partial F_i}{\partial T} - \frac{\partial P_{r,i}}{\partial T}c_i\right) \quad , \tag{C.127}$$

(C.128)

and for the state parameter derivative:

$$\frac{\partial c_i}{\partial V} = \frac{1}{P_{r,i}+1}\left(\frac{\partial F_i}{\partial V} - \frac{\partial P_{r,i}}{\partial V}c_i\right) \quad \text{for CONP,} \tag{C.129a}$$

$$\frac{\partial c_i}{\partial P} = \frac{1}{P_{r,i}+1}\left(\frac{\partial F_i}{\partial P} - \frac{\partial P_{r,i}}{\partial P}c_i\right) \quad \text{for CONV,} \tag{C.129b}$$

and finally, the molar derivative:

$$\frac{\partial c_i}{\partial n_j} = \frac{1}{P_{r,i}+1}\left(\frac{\partial F_i}{\partial n_j} - \frac{\partial P_{r,i}}{\partial n_j}c_i\right) \quad . \tag{C.130}$$

**Reduced pressure**

Next the reduced pressure derivatives in Eqs. (C.124) to (C.129) will be evaluated; these depend on the form of the third-body efficiency used in Eq. (C.38), thus we will first work out the derivatives for each third-body efficiency form, and then develop a generalized form for compact representation. For a third-body concentration based upon the entire mixture, (i.e., Eq. (C.34)) the temperature derivative is:

$$\frac{\partial P_{r,i}}{\partial T} = \frac{P_{r,i}}{T}\left(\beta_0 - \beta_\infty + \frac{E_{a,0}}{T\mathcal{R}} - \frac{E_{a,\infty}}{T\mathcal{R}}\right) - \frac{[C]k_{0,i}\alpha_{N_{\text{sp}},i}}{Tk_{\infty,i}} \quad , \tag{C.131}$$

and the state parameter derivatives are:

$$\frac{\partial P_{r,i}}{\partial V} = -\frac{P_{r,i}}{V} + \frac{[C]k_{0,i}\alpha_{N_{\text{sp}},i}}{Vk_{\infty,i}} \quad \text{for CONP,} \tag{C.132a}$$

$$\frac{\partial P_{r,i}}{\partial P} = \frac{k_{0,i}\alpha_{N_{\text{sp}},i}}{Tk_{\infty,i}\mathcal{R}} \quad \text{for CONV,} \tag{C.132b}$$

and finally, the molar derivative is given by:

$$\frac{\partial P_{r,i}}{\partial n_j} = \frac{k_{0,i}\left(-\alpha_{N_{\text{sp}},i} + \alpha_{j,i}\right)}{k_{\infty,i}V} \quad . \tag{C.133}$$

For a unity third-body concentration (i.e., all $\alpha_{k,i} = 1$):

$$\frac{\partial P_{r,i}}{\partial T} = \frac{P_{r,i}}{T}\left(\beta_0 - \beta_\infty - 1 + \frac{E_{a,0}}{T\mathcal{R}} - \frac{E_{a,\infty}}{T\mathcal{R}}\right) \quad , \tag{C.134}$$

and the state parameter derivatives are:

$$\frac{\partial P_{r,i}}{\partial V} = 0 \quad \text{for CONP,} \tag{C.135a}$$

$$\frac{\partial P_{r,i}}{\partial P} = \frac{k_{0,i}}{k_{\infty,i}}\frac{1}{\mathcal{R}T} \quad \text{for CONV,} \tag{C.135b}$$

and the molar derivative is:

$$\frac{\partial P_{r,i}}{\partial n_j} = 0 \quad . \tag{C.136}$$

Finally, for species-based third-body (i.e., the species $m$ alone acts as the third body):

$$\frac{\partial P_{r,i}}{\partial T} = \frac{P_{r,i}}{T}\left(\beta_0 - \beta_\infty + \frac{E_{a,0}}{T\mathcal{R}} - \frac{E_{a,\infty}}{T\mathcal{R}}\right) - \frac{[C]k_{0,i}\delta_{N_{\text{sp}}m}}{Tk_{\infty,i}} \quad , \tag{C.137}$$

for the state parameter derivative:

$$\frac{\partial P_{r,i}}{\partial V} = -\frac{P_{r,i}}{V} + \frac{[C]k_{0,i}\delta_{N_{\mathrm{sp}}m}}{Vk_{\infty,i}} \qquad \text{for CONP,} \tag{C.138a}$$

$$\frac{\partial P_{r,i}}{\partial P} = \frac{k_{0,i}\delta_{N_{\mathrm{sp}}m}}{Tk_{\infty,i}\mathcal{R}} \qquad \text{for CONV,} \tag{C.138b}$$

and the molar derivative:

$$\frac{\partial P_{r,i}}{\partial n_j} = \frac{k_{0,i}}{Vk_{\infty,i}}\left(-\delta_{N_{\mathrm{sp}}m} + \delta_{jm}\right) \quad . \tag{C.139}$$

Equations (C.131) to (C.138) may be recast in a more generalized form using temporary variables $\Theta_{P_{r,i},\partial\ldots}$ and $\bar{\theta}_{P_{r,i},\partial\ldots}$—where, for example, $\Theta_{P_{r,i},\partial T}$ corresponds to part of the reduced-pressure derivative with respect to temperature—to replace terms in each individual form of the reduced-pressure derivatives:

$$\frac{\partial P_{r,i}}{\partial T} = P_{r,i}\Theta_{P_{r,i},\partial T} + \bar{\theta}_{P_{r,i},\partial T} \quad , \tag{C.140}$$

and the state parameter:

$$\frac{\partial P_{r,i}}{\partial V} = P_{r,i}\Theta_{P_{r,i},\partial V} + \bar{\theta}_{P_{r,i},\partial V} \qquad \text{for CONP,} \tag{C.141a}$$

$$\frac{\partial P_{r,i}}{\partial P} = \bar{\theta}_{P_{r,i},\partial P} \qquad \text{for CONV,} \tag{C.141b}$$

and the molar derivative:

$$\frac{\partial P_{r,i}}{\partial n_j} = \frac{k_{0,i}\bar{\theta}_{P_{r,i},\partial n_j}}{Vk_{\infty,i}} \quad . \tag{C.142}$$

with:

$$\Theta_{P_{r,i},\partial T} = \begin{cases} \dfrac{1}{T}\left(\beta_0 - \beta_\infty + \dfrac{E_{a,0}}{T\mathcal{R}} - \dfrac{E_{a,\infty}}{T\mathcal{R}}\right) & \text{if mix,} \\ \dfrac{1}{T}\left(\beta_0 - \beta_\infty - 1 + \dfrac{E_{a,0}}{T\mathcal{R}} - \dfrac{E_{a,\infty}}{T\mathcal{R}}\right) & \text{if unity,} \\ \dfrac{1}{T}\left(\beta_0 - \beta_\infty + \dfrac{E_{a,0}}{T\mathcal{R}} - \dfrac{E_{a,\infty}}{T\mathcal{R}}\right) & \text{if species,} \end{cases} \tag{C.143}$$

$$\bar{\theta}_{P_{r,i},\partial T} = \begin{cases} -\dfrac{[C]k_{0,i}\alpha_{N_{\mathrm{sp}},i}}{Tk_{\infty,i}} & \text{if mix,} \\ 0 & \text{if unity,} \\ -\dfrac{[C]k_{0,i}\delta_{N_{\mathrm{sp}}m}}{Tk_{\infty,i}} & \text{if species,} \end{cases} \tag{C.144}$$

$$\Theta_{P_{r,i},\partial V} = \begin{cases} -\dfrac{1}{V} & \text{if mix,} \\ 0 & \text{if unity,} \\ -\dfrac{1}{V} & \text{if species,} \end{cases} \tag{C.145}$$

$$\bar{\theta}_{P_{r,i},\partial V} = \begin{cases} \dfrac{[C]k_{0,i}\alpha_{N_{\text{sp}},i}}{Vk_{\infty,i}} & \text{if mix,} \\ 0 & \text{if unity,} \\ \dfrac{[C]k_{0,i}\delta_{N_{\text{sp}}m}}{Vk_{\infty,i}} & \text{if species,} \end{cases} \tag{C.146}$$

$$\bar{\theta}_{P_{r,i},\partial P} = \begin{cases} \dfrac{k_{0,i}\alpha_{N_{\text{sp}},i}}{Tk_{\infty,i}\mathcal{R}} & \text{if mix,} \\ \dfrac{k_{0,i}}{Tk_{\infty,i}\mathcal{R}} & \text{if unity,} \\ \dfrac{k_{0,i}\delta_{N_{\text{sp}}m}}{Tk_{\infty,i}\mathcal{R}} & \text{if species,} \end{cases} \tag{C.147}$$

$$\bar{\theta}_{P_{r,i},\partial n_j} = \begin{cases} -\alpha_{N_{\text{sp}},i} + \alpha_{j,i} & \text{if mix,} \\ 0 & \text{if unity,} \\ -\delta_{N_{\text{sp}}m} + \delta_{jm} & \text{if species.} \end{cases} \tag{C.148}$$

**Falloff blending factor**

Derivatives of the falloff blending factor—except for all but the Lindemann falloff blending factor, which is not a function of the reduced pressure—rely upon the reduced pressure derivatives developed in Appendix C.3.6. First, the blending factor for each falloff reaction type will be differentiated along with any subcomponents (e.g., the $F_{cent}$ term for Troe falloff reactions). Next, the reduced pressured derivatives given by Eqs. (C.140) to (C.142) will be substituted in and finally a generalized form for the falloff blending factor derivatives will be determined.
For Lindemann falloff reactions:

$$\frac{\partial F_i}{\partial T} = 0 \qquad , \tag{C.149}$$

$$\tag{C.150}$$

and, for the state parameter:

$$\frac{\partial F_i}{\partial V} = 0 \qquad \text{for CONP,} \tag{C.151a}$$

$$\frac{\partial F_i}{\partial P} = 0 \qquad \text{for CONV,} \tag{C.151b}$$

and the molar derivative:

$$\frac{\partial F_i}{\partial n_j} = 0 \quad . \tag{C.152}$$

For Troe falloff reactions:

$$\frac{\partial F_i}{\partial T} = \frac{\partial F_i}{\partial F_{cent}} \frac{\mathrm{d}F_{cent}}{\mathrm{d}T} + \frac{\partial F_i}{\partial P_{r,i}} \frac{\partial P_{r,i}}{\partial T} \quad , \tag{C.153}$$

and the state parameter derivative:

$$\frac{\partial F_i}{\partial V} = \frac{\partial F_i}{\partial P_{r,i}} \frac{\partial P_{r,i}}{\partial V} \qquad \text{for CONP,} \tag{C.154a}$$

$$\frac{\partial F_i}{\partial P} = \frac{\partial F_i}{\partial P_{r,i}} \frac{\partial P_{r,i}}{\partial P} \qquad \text{for CONV,} \tag{C.154b}$$

and:

$$\frac{\partial F_i}{\partial n_j} = \frac{\partial F_i}{\partial P_{r,i}} \frac{\partial P_{r,i}}{\partial n_j} \quad . \tag{C.155}$$

The Troe falloff blending factor derivative with respect to $F_{cent}$ is:

$$\frac{\partial F_i}{\partial F_{cent}} = \frac{F_i}{\frac{A_{Troe}^2}{B_{Troe}^2} + 1} \left( \frac{2 A_{Troe} \log\left(F_{cent}\right)}{B_{Troe}^2 \left(\frac{A_{Troe}^2}{B_{Troe}^2} + 1\right)} \left( \frac{A_{Troe}}{B_{Troe}} \frac{\partial B_{Troe}}{\partial F_{cent}} - \frac{\partial A_{Troe}}{\partial F_{cent}} \right) + \frac{1}{F_{cent}} \right) \quad , \tag{C.156}$$

and:

$$\frac{\mathrm{d}F_{cent}}{\mathrm{d}T} = \frac{a}{T^*} \exp\left(-\frac{T}{T^*}\right) - \frac{\exp\left(-\frac{T}{T^{***}}\right)}{T^{***}} \left(-a + 1\right) + \frac{T^{**}}{T^2} \exp\left(-\frac{T^{**}}{T}\right) \quad , \tag{C.157}$$

and:

$$\frac{\partial F_i}{\partial P_{r,i}} = \frac{2 F_i A_{Troe} \log\left(F_{cent}\right)}{B_{Troe}^2 \left(\frac{A_{Troe}^2}{B_{Troe}^2} + 1\right)^2} \left( \frac{A_{Troe}}{B_{Troe}} \frac{\partial B_{Troe}}{\partial P_{r,i}} - \frac{\partial A_{Troe}}{\partial P_{r,i}} \right) \quad . \tag{C.158}$$

Finally, the derivatives of $A_{Troe}$ and $B_{Troe}$ are:

$$\frac{\partial A_{Troe}}{\partial F_{cent}} = -\frac{0.67}{F_{cent} \ln\left(10\right)} \quad , \tag{C.159}$$

$$\frac{\partial B_{Troe}}{\partial F_{cent}} = -\frac{1.1762}{F_{cent}\ln(10)} \quad , \tag{C.160}$$

$$\frac{\partial A_{Troe}}{\partial P_{r,i}} = \frac{1}{P_{r,i}\ln(10)} \quad , \tag{C.161}$$

and:

$$\frac{\partial B_{Troe}}{\partial P_{r,i}} = -\frac{0.14}{P_{r,i}\ln(10)} \quad . \tag{C.162}$$

$$\tag{C.163}$$

Combining Eqs. (C.159) to (C.162) with Eqs. (C.156) and (C.158) yields:

$$\frac{\partial F_i}{\partial F_{cent}} = -\frac{F_i B_{Troe}}{F_{cent}\left(A_{Troe}^2 + B_{Troe}^2\right)^2 \ln(10)} \left(2A_{Troe}\left(1.1762A_{Troe} - 0.67B_{Troe}\right)\log\left(F_{cent}\right) - B_{Troe}\left(A_{Troe}^2 + B_{Troe}^2\right)\ln(10)\right) \quad , \tag{C.164}$$

and:

$$\frac{\partial F_i}{\partial P_{r,i}} = -\frac{2F_i A_{Troe}\left(\frac{0.14A_{Troe}}{B_{Troe}} + 1\right)\log\left(F_{cent}\right)}{B_{Troe}^2 P_{r,i}\left(\frac{A_{Troe}^2}{B_{Troe}^2} + 1\right)^2 \ln(10)} \tag{C.165}$$

Equations (C.157), (C.164) and (C.165) can then be substituted into Eqs. (C.153) to (C.155) to obtain a final form for the Troe-falloff blending function derivatives, which will be given as a generalized form at the end of this section.

For SRI falloff reactions, the temperature derivative of the falloff blending factor is:

$$\frac{\partial F_i}{\partial T} = F_i\left(\frac{X\left(-\frac{\exp\left(-\frac{T}{c}\right)}{c} + \frac{ab}{T^2}\exp\left(-\frac{b}{T}\right)\right)}{a\exp\left(-\frac{b}{T}\right) + \exp\left(-\frac{T}{c}\right)} + \frac{\partial P_{r,i}}{\partial T}\frac{\mathrm{d}X}{\mathrm{d}P_{r,i}}\log\left[a\exp\left(-\frac{b}{T}\right) + \exp\left(-\frac{T}{c}\right)\right] + \frac{e}{T}\right) \quad , \tag{C.166}$$

and the state parameter derivatives:

$$\frac{\partial F_i}{\partial V} = F_i\frac{\partial P_{r,i}}{\partial V}\frac{\mathrm{d}X}{\mathrm{d}P_{r,i}}\log\left(a\exp\left(-\frac{b}{T}\right) + \exp\left(-\frac{T}{c}\right)\right) \qquad \text{for CONP,} \tag{C.167a}$$

$$\frac{\partial F_i}{\partial P} = F_i\frac{\partial P_{r,i}}{\partial P}\frac{\mathrm{d}X}{\mathrm{d}P_{r,i}}\log\left(a\exp\left(-\frac{b}{T}\right) + \exp\left(-\frac{T}{c}\right)\right) \qquad \text{for CONV,} \tag{C.167b}$$

and finally, with respect to species moles:

$$\frac{\partial F_i}{\partial n_j} = F_i \frac{\partial P_{r,i}}{\partial n_j} \frac{\mathrm{d}X}{\mathrm{d}P_{r,i}} \log\left(a \exp\left(-\frac{b}{T}\right) + \exp\left(-\frac{T}{c}\right)\right) \quad . \tag{C.168}$$

The derivatives of the SRI exponent $X$ are:

$$\frac{\mathrm{d}X}{\mathrm{d}P_{r,i}} = -\frac{2X^2 \log\left(P_{r,i}\right)}{P_{r,i} \ln^2\left(10\right)} \quad , \tag{C.169}$$

$$\frac{\partial X}{\partial n_j} = \frac{\partial P_{r,i}}{\partial n_j} \frac{\mathrm{d}X}{\mathrm{d}P_{r,i}} \quad . \tag{C.170}$$

As with the reduced-pressure derivatives, Eqs. (C.149), (C.151) to (C.155) and (C.166) to (C.168) may be represented in a general form via introduction of temporary variables $\Theta_{F_i,\partial\ldots}$, giving for the temperature derivative:

$$\frac{\partial F_i}{\partial T} = F_i \Theta_{F_i,\partial T} \quad , \tag{C.171}$$

for the state parameter derivatives:

$$\frac{\partial F_i}{\partial V} = F_i \Theta_{F_i,\partial V} \qquad \text{for CONP,} \tag{C.172a}$$

$$\frac{\partial F_i}{\partial P} = F_i \Theta_{F_i,\partial P} \qquad \text{for CONV,} \tag{C.172b}$$

and finally, for the molar derivative:

$$\frac{\partial F_i}{\partial n_j} = \frac{F_i k_{0,i} \Theta_{F_i,\partial n_j}}{V k_{\infty,i}} \bar{\theta}_{P_{r,i},\partial n_j} \quad . \tag{C.173}$$

For Lindemann falloff reactions:

$$\Theta_{F_i,\partial T} = 0 \quad , \tag{C.174}$$

and, for the state parameter derivatives:

$$\Theta_{F_i,\partial V} = 0 \quad , \tag{C.175a}$$

$$\Theta_{F_i,\partial P} = 0 \quad , \tag{C.175b}$$

and for the molar derivative:

$$\Theta_{F_i,\partial n_j} = 0 \quad . \tag{C.176}$$

For Troe falloff reactions:

$$\Theta_{F_i,\partial T} = -\frac{B_{Troe}}{F_{cent}P_{r,i}\left(A_{Troe}^2 + B_{Troe}^2\right)^2 \ln(10)} \times \Bigg( 2A_{Troe}F_{cent}\left(0.14A_{Troe} + B_{Troe}\right) \times \left(P_{r,i}\Theta_{P_{r,i},\partial T} + \bar{\theta}_{P_{r,i},\partial T}\right)\log\left(F_{cent}\right) + P_{r,i}\frac{\mathrm{d}F_{cent}}{\mathrm{d}T} \times \Big( 2A_{Troe}\left(1.1762A_{Troe} - 0.67B_{Troe}\right)\log\left(F_{cent}\right) - B_{Troe}\left(A_{Troe}^2 + B_{Troe}^2\right)\ln(10)\Big)\Bigg) \quad , \tag{C.177}$$

and for the state parameter derivatives:

$$\Theta_{F_i,\partial V} = -\frac{2A_{Troe}B_{Troe}\log\left(F_{cent}\right)}{P_{r,i}\left(A_{Troe}^2 + B_{Troe}^2\right)^2 \ln(10)} \times \Big( 0.14A_{Troe} + B_{Troe}\Big)\left(P_{r,i}\Theta_{P_{r,i},\partial V} + \bar{\theta}_{P_{r,i},\partial V}\right) \quad , \tag{C.178a}$$

$$\Theta_{F_i,\partial P} = -\frac{2A_{Troe}B_{Troe}\bar{\theta}_{P_{r,i},\partial P}\left(0.14A_{Troe} + B_{Troe}\right)\log\left(F_{cent}\right)}{P_{r,i}\left(A_{Troe}^2 + B_{Troe}^2\right)^2 \ln(10)} \, , \tag{C.178b}$$

and the molar derivative:

$$\Theta_{F_i,\partial n_j} = -\frac{2A_{Troe}B_{Troe}\left(0.14A_{Troe} + B_{Troe}\right)\log\left(F_{cent}\right)}{P_{r,i}\left(A_{Troe}^2 + B_{Troe}^2\right)^2 \ln(10)} \quad . \tag{C.179}$$

where $\Theta_{P_{r,i},\partial\ldots}$ and $\bar{\theta}_{P_{r,i},\partial\ldots}$ are given by Eqs. (C.143) to (C.148) respectively.

Finally, for SRI falloff reactions:

$$\Theta_{F_i,\partial T} = -\frac{X\left(\frac{\exp\left(-\frac{T}{c}\right)}{c} - \frac{ab}{T^2}\exp\left(-\frac{b}{T}\right)\right)}{a\exp\left(-\frac{b}{T}\right) + \exp\left(-\frac{T}{c}\right)} + \frac{e}{T} - \Bigg( \frac{2X^2\log\left(a\exp\left(-\frac{b}{T}\right) + \exp\left(-\frac{T}{c}\right)\right)}{P_{r,i}\ln^2(10)} \times \left(P_{r,i}\Theta_{P_{r,i},\partial T} + \bar{\theta}_{P_{r,i},\partial T}\right)\log\left(P_{r,i}\right)\Bigg) \quad , \tag{C.180}$$

and:

$$\Theta_{F_i,\partial V} = -\frac{2X^2\log\left(P_{r,i}\right)}{P_{r,i}\ln^2(10)}\left(P_{r,i}\Theta_{P_{r,i},\partial V} + \bar{\theta}_{P_{r,i},\partial V}\right)\log\Bigg( \left(a\exp\left(\frac{T}{c}\right) + \exp\left(\frac{b}{T}\right)\right)\exp\left(-\frac{T}{c} - \frac{b}{T}\right)\Bigg) \quad , \tag{C.181a}$$

$$\Theta_{F_i,\partial P} = -\frac{2X^2\bar{\theta}_{P_{r,i},\partial P}\log\left(P_{r,i}\right)}{P_{r,i}\ln^2(10)}\log\left(a\exp\left(-\frac{b}{T}\right) + \exp\left(-\frac{T}{c}\right)\right) \, , \tag{C.181b}$$

and finally, for the molar derivative:

$$\Theta_{F_i,\partial n_j} = -\frac{2X^2 \log\left(a\exp\left(-\frac{b}{T}\right)+\exp\left(-\frac{T}{c}\right)\right)}{P_{r,i}\ln^2(10)}\log\left(P_{r,i}\right) \quad . \tag{C.182}$$

**Unimolecular/recombination falloff reactions (temporary product form)**

Using the temporary products developed in Appendix C.3.6, the falloff reaction derivatives developed in Eqs. (C.124) to (C.126) may be simplified to:

$$\frac{\partial c_i}{\partial T} = \frac{F_i\bar{\theta}_{P_{r,i},\partial T}}{P_{r,i}+1} + \left(-\frac{P_{r,i}\Theta_{P_{r,i},\partial T}}{P_{r,i}+1} + \Theta_{F_i,\partial T} + \Theta_{P_{r,i},\partial T} - \frac{\bar{\theta}_{P_{r,i},\partial T}}{P_{r,i}+1}\right)c_i \quad , \tag{C.183}$$

and, for the state parameter derivatives:

$$\frac{\partial c_i}{\partial V} = \frac{F_i\bar{\theta}_{P_{r,i},\partial V}}{P_{r,i}+1} + \left(-\frac{P_{r,i}\Theta_{P_{r,i},\partial V}}{P_{r,i}+1} + \Theta_{F_i,\partial V} + \Theta_{P_{r,i},\partial V} - \frac{\bar{\theta}_{P_{r,i},\partial V}}{P_{r,i}+1}\right)c_i \quad \text{for CONP,} \tag{C.184a}$$

$$\frac{\partial c_i}{\partial P} = \frac{F_i\bar{\theta}_{P_{r,i},\partial P}}{P_{r,i}+1} + \left(\Theta_{F_i,\partial P} - \frac{\bar{\theta}_{P_{r,i},\partial P}}{P_{r,i}+1}\right)c_i \quad \text{for CONV,} \tag{C.184b}$$

and the molar derivative:

$$\frac{\partial c_i}{\partial n_j} = \frac{k_{0,i}\bar{\theta}_{P_{r,i},\partial n_j}}{Vk_{\infty,i}\left(P_{r,i}+1\right)}\left(F_i\left(P_{r,i}\Theta_{F_i,\partial n_j}+1\right)-c_i\right) \quad . \tag{C.185}$$

The temporary products in Eqs. (C.183) to (C.185) are given by Eqs. (C.143) to (C.148) and Eqs. (C.174) to (C.182).

**Chemically-activated bimolecular reactions (temporary product form)**

Similarly, the chemically-activated reaction derivatives in Eqs. (C.127), (C.129) and (C.130) may be written as:

$$\frac{\partial c_i}{\partial T} = \left(-\frac{P_{r,i}\Theta_{P_{r,i},\partial T}}{P_{r,i}+1} + \Theta_{F_i,\partial T} - \frac{\bar{\theta}_{P_{r,i},\partial T}}{P_{r,i}+1}\right)c_i \quad , \tag{C.186}$$

and the state parameter derivatives:

$$\frac{\partial c_i}{\partial V} = \left(-\frac{P_{r,i}\Theta_{P_{r,i},\partial V}}{P_{r,i}+1} + \Theta_{F_i,\partial V} - \frac{\bar{\theta}_{P_{r,i},\partial V}}{P_{r,i}+1}\right)c_i \quad \text{for CONP} \tag{C.187a}$$

$$\frac{\partial c_i}{\partial P} = \left(\Theta_{F_i,\partial P} - \frac{\bar{\theta}_{P_{r,i},\partial P}}{P_{r,i}+1}\right)c_i \quad \text{for CONV} \tag{C.187b}$$

and the molar derivative:

$$\frac{\partial c_i}{\partial n_j} = \frac{k_{0,i}\bar{\theta}_{P_{r,i},\partial n_j}\left(F_i\Theta_{F_i,\partial n_j} - c_i\right)}{k_{\infty,i}V\left(P_{r,i}+1\right)} \quad . \tag{C.188}$$

The temporary products are the same as those detailed in Appendix C.3.6.

## C.4 Final Jacobian form

Here we give a summary of the Jacobian derivations in the previous sections, as a complete reference guide to the equations evaluated by `pyJac`.

### C.4.1 Temperature derivatives

In this section, the derivatives of all source terms with respect to the temperature are considered.

#### Temperature source term derivatives

Starting from Eq. (C.59), Eq. (C.71) is substituted in to yield:

$$\begin{aligned}\mathcal{J}_{1,1} = \frac{1}{\sum_{k=1}^{N_{\text{sp}}}[C]_k C_{p,k}}\Bigg[&\frac{\mathrm{d}T}{\mathrm{d}t}\sum_{k=1}^{N_{\text{sp}}}\left(-\frac{\mathrm{d}C_{p,k}}{\mathrm{d}T}+\frac{C_{p,N_{\text{sp}}}}{T}\right)[C]_k + \\ &\sum_{k=1}^{N_{\text{sp}}-1}\Bigg(\left(-H_k+\frac{W_k H_{N_{\text{sp}}}}{W_{N_{\text{sp}}}}\right)\frac{\mathcal{J}_{k+2,1}}{V} + \\ &\left(-C_{p,k}+\frac{W_k C_{p,k}}{W_{N_{\text{sp}}}}\right)\dot{\omega}_k\Bigg)\Bigg] \qquad \text{for CONP,}\end{aligned} \tag{C.189a}$$

and similarly:

$$\begin{aligned}\mathcal{J}_{1,1} = \frac{1}{\sum_{k=1}^{N_{\text{sp}}}[C]_k C_{v,k}}\Bigg[&\frac{\mathrm{d}T}{\mathrm{d}t}\sum_{k=1}^{N_{\text{sp}}}\left(-\frac{\mathrm{d}C_{v,k}}{\mathrm{d}T}+\frac{C_{v,N_{\text{sp}}}}{T}\right)[C]_k + \\ &\sum_{k=1}^{N_{\text{sp}}-1}\Bigg(\left(-U_k+\frac{W_k U_{N_{\text{sp}}}}{W_{N_{\text{sp}}}}\right)\frac{\mathcal{J}_{k+2,1}}{V} + \\ &\left(-C_{v,k}+\frac{W_k C_{v,N_{\text{sp}}}}{W_{N_{\text{sp}}}}\right)\dot{\omega}_k\Bigg)\Bigg] \qquad \text{for CONV.}\end{aligned} \tag{C.189b}$$

#### State parameter source term derivatives

Similarly, Eqs. (C.52) and (C.71) are substituted into Eq. (C.68), giving:

$$\begin{aligned}\mathcal{J}_{2,1} &= \frac{\partial}{\partial T}\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{\partial \dot{V}}{\partial T} \\ &= \frac{V}{[C]}\sum_{k=1}^{N_{\text{sp}}-1}\left(1-\frac{W_k}{W_{N_{\text{sp}}}}\right)\left(\frac{\mathcal{J}_{k+2,1}}{V}+\frac{\dot{\omega}_k}{T}\right)\end{aligned}$$

$$+\frac{V}{T}\left(\mathcal{J}_{1,1}-\frac{1}{T}\frac{\mathrm{d}T}{\mathrm{d}t}\right) \qquad \text{for CONP,} \tag{C.190a}$$

and:

$$\begin{aligned}\mathcal{J}_{2,1} &= \frac{\partial}{\partial T}\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{\partial \dot{P}}{\partial T} \\ &= \mathcal{R}\sum_{k=1}^{N_{\mathrm{sp}}-1}\left(1-\frac{W_k}{W_{N_{\mathrm{sp}}}}\right)\left(T\frac{\mathcal{J}_{k+2,1}}{V}+\dot{\omega}_k\right) \\ &\quad +\frac{P}{T}\left(\mathcal{J}_{1,1}-\frac{1}{T}\frac{\mathrm{d}T}{\mathrm{d}t}\right) \qquad \text{for CONV.}\end{aligned} \tag{C.190b}$$

**Molar source term derivatives**

From Eq. (C.71) and Eq. (C.74), a general form for the molar source term derivative with respect to temperature is obtained:

$$\mathcal{J}_{k+2,1} = V\sum_{i=1}^{N_{\mathrm{reac}}}\left(\nu_{k,i}R_i\frac{\partial c_i}{\partial T}+\nu_{k,i}\frac{\partial R_i}{\partial T}c_i\right) \quad , \tag{C.191}$$

where the rate of progress derivatives are given by Eqs. (C.89), (C.100) and (C.107) for Arrhenius-based—i.e., elementary, third-body and falloff/chemically-activated—reactions, P-Log reactions and Chebyshev reactions respectively. The temperature derivatives of the pressure modification term are found in Eqs. (C.115), (C.118) and (C.121) for third-body reactions, and Eqs. (C.183) and (C.186) for falloff/chemically-activated reactions respectively.

### C.4.2 State parameter derivatives

In this section, the derivatives of all source terms with respect to the state parameter are considered.

**Temperature source term derivatives**

Similar to Appendix C.4.1, Eq. (C.72) is substituted into Eq. (C.62) to yield:

$$\begin{aligned}\mathcal{J}_{1,2} = \frac{1}{\sum_{k=1}^{N_{\mathrm{sp}}}V[C]_kC_{p,k}}\Bigg[ & \\ -\sum_{k=1}^{N_{\mathrm{sp}}-1}\left(H_k-\frac{W_kH_{N_{\mathrm{sp}}}}{W_{N_{\mathrm{sp}}}}\right)\left(\mathcal{J}_{k+2,2}-\dot{\omega}_k\right)+ & \\ \frac{\mathrm{d}T}{\mathrm{d}t}\sum_{k=1}^{N_{\mathrm{sp}}-1}\left(-C_{p,N_{\mathrm{sp}}}+C_{p,k}\right)[C]_k\Bigg] & \qquad \text{for CONP,}\end{aligned} \tag{C.192a}$$

$$\mathcal{J}_{1,2} = \frac{1}{\sum_{k=1}^{N_{\mathrm{sp}}}[C]_kC_{v,k}}\Bigg[$$

$$- \sum_{k=1}^{N_{\text{sp}}-1} \left( U_k - \frac{W_k U_{N_{\text{sp}}}}{W_{N_{\text{sp}}}} \right) \frac{\mathcal{J}_{k+2,2}}{V} - \frac{C_{v,N_{\text{sp}}}}{T\mathcal{R}} \frac{\mathrm{d}T}{\mathrm{d}t} \Bigg] \qquad \text{for CONV.} \tag{C.192b}$$

**State parameter source term derivatives**

Next, Eq. (C.69) is simplified via Eqs. (C.60) and (C.72) to give:

$$\mathcal{J}_{2,2} = \frac{1}{[C]} \sum_{k=1}^{N_{\text{sp}}-1} \left( 1 - \frac{W_k}{W_{N_{\text{sp}}}} \right) \mathcal{J}_{k+2,2} + \frac{1}{T} \left( V \mathcal{J}_{1,2} + \dot{T} \right) \qquad \text{for CONP,} \tag{C.193a}$$

and:

$$\mathcal{J}_{2,2} = \frac{T\mathcal{R}}{V} \sum_{k=1}^{N_{\text{sp}}-1} \left( 1 - \frac{W_k}{W_{N_{\text{sp}}}} \right) \mathcal{J}_{k+2,2} + \frac{1}{T} \left( P \mathcal{J}_{1,2} + \dot{T} \right) \qquad \text{for CONV.} \tag{C.193b}$$

**Molar source term derivatives**

To obtain a final form for the molar source term derivative with respect to the state parameter, Eqs. (C.72) and (C.75) are combined:

$$\mathcal{J}_{k+2,2} = V \sum_{i=1}^{N_{\text{reac}}} \left( \nu_{k,i} R_i \frac{\partial c_i}{\partial V} + \nu_{k,i} \frac{\partial R_i}{\partial V} c_i \right) + \dot{\omega}_k \qquad \text{for CONP,} \tag{C.194a}$$

$$\mathcal{J}_{k+2,2} = V \sum_{i=1}^{N_{\text{reac}}} \left( \nu_{k,i} R_i \frac{\partial c_i}{\partial P} + \nu_{k,i} \frac{\partial R_i}{\partial P} c_i \right) \qquad \text{for CONV.} \tag{C.194b}$$

The derivative of the net rate of progress with respect to the state parameter is given by Eq. (C.92) for arrhenius-based reactions. For P-Log and Chebyshev reactions in a CONV system Eqs. (C.102) and (C.111) are used, while Eq. (C.92) is used for a CONP system. The pressure modification term derivatives are given by Eqs. (C.116), (C.119) and (C.122) for third-body enhanced reactions, and Eqs. (C.184) and (C.187) for falloff/chemically-activated reactions respectively.

### C.4.3 Molar derivatives

Finally, this section details the Jacobian entries corresponding to derivatives with respect to the species moles.

**Temperature source term derivatives**

The temperature source term derivative with respect to the moles of species $j$, Eq. (C.67), is combined with Eq. (C.73), giving:

$$\mathcal{J}_{1,j+2} = \frac{1}{\sum_{k=1}^{N_{\text{sp}}} V[C]_k C_{p,k}} \Bigg[ - \sum_{k=1}^{N_{\text{sp}}-1} \left( H_k - \frac{W_k H_{N_{\text{sp}}}}{W_{N_{\text{sp}}}} \right) \mathcal{J}_{k+2,j+2} - \frac{\mathrm{d}T}{\mathrm{d}t} \left( -C_{p,N_{\text{sp}}} + C_{p_j} \right) \Bigg] \qquad \text{for CONP,} \tag{C.195a}$$

and:

$$\mathcal{J}_{1,j+2} = \frac{1}{\sum_{k=1}^{N_{\text{sp}}} V[C]_k C_{v,k}} \Bigg[ - \sum_{k=1}^{N_{\text{sp}}-1} \left( U_k - \frac{W_k U_{N_{\text{sp}}}}{W_{N_{\text{sp}}}} \right) \mathcal{J}_{k+2,j+2} - \frac{\mathrm{d}T}{\mathrm{d}t} \left( -C_{v,N_{\text{sp}}} + C_{vj} \right) \Bigg] \qquad \text{for CONV.} \tag{C.195b}$$

**State parameter derivatives**

From Eqs. (C.63), (C.70) and (C.73), the final form of the state parameter source term derivative with respect to the moles of species $j$ is:

$$\mathcal{J}_{2,j+2} = \frac{1}{[C]} \sum_{k=1}^{N_{\text{sp}}-1} \left( 1 - \frac{W_k}{W_{N_{\text{sp}}}} \right) \mathcal{J}_{k+2,j+2} + \frac{V}{T} \mathcal{J}_{1,j+2} \qquad \text{for CONP,} \tag{C.196a}$$

and:

$$\mathcal{J}_{2,j+2} = \frac{T\mathcal{R}}{V} \sum_{k=1}^{N_{\text{sp}}-1} \left( 1 - \frac{W_k}{W_{N_{\text{sp}}}} \right) \mathcal{J}_{k+2,j+2} + \frac{P}{T} \mathcal{J}_{2,j+2} \qquad \text{for CONV.} \tag{C.196b}$$

**Molar source term derivatives**

Finally, Eqs. (C.73) and (C.76) are combined yielding:

$$\mathcal{J}_{k+2,j+2} = V \sum_{i=1}^{N_{\text{reac}}} \left( \nu_{k,i} R_i \frac{\partial c_i}{\partial n_j} + \nu_{k,i} \frac{\partial R_i}{\partial n_j} c_i \right) \quad . \tag{C.197}$$

As `pyJac` evaluates these derivative entries by looping over the reactions $i$ in the model, Eq. (C.197) is converted to an update form:

$$\mathcal{J}_{k+2,j+2} \mathrel{+}= V\left(\nu_{k,i} R_i \frac{\partial c_i}{\partial n_j} + \nu_{k,i} \frac{\partial R_i}{\partial n_j} c_i\right) \qquad k = 1, \ldots, N_{\text{sp}} - 1 \tag{C.198}$$

The molar derivatives of the net rate of progress are given by Eq. (C.97), and the pressure modification term by Eqs. (C.117), (C.120) and (C.123) for third-body enhanced reactions, and Eqs. (C.185) and (C.188) for falloff and chemically-activated reactions, respectively.

# Appendix D

# Supplemental Materials for "Using SIMD and SIMT vectorization to evaluate sparse chemical kinetic Jacobian matrices and thermochemical source terms"

## D.1 Availability of material

The results for this article were obtained using `pyJac v2.0.0b0` [200]. The most recent version of `pyJac` can be found at its GitHub repository: https://github.com/SLACKHA/pyJac. All figures, and the data and plotting scripts necessary to reproduce them, are available openly under the CC-BY license [201].

## D.2 Jacobian error statistics per test platform

This section gives more detail on the results presented in Section 3.3.3, breaking down the reported error statistics per test platform/language. The error of the Intel OpenCL runtime is presented in Table D.1, the Portable OpenCL (POCL) runtime in Table D.2, OpenMP in Table D.3, and the Nvidia OpenCL runtime in Table D.4. POCL and OpenMP tend to have the smallest error norms, while Nvidia tends to have the largest; in particular the stringent

filtered error norm $E_{\mathcal{C}=10^{20}}$ is two orders of magnitude larger for the Nvidia runtime than the other test platforms with the $H_2$/CO and USC-Mech II models.

| Model | $E_{\mathcal{L}}$ | $E_{\mathcal{C}=10^{20}}$ | $E_{\mathcal{C}=10^{15}}$ |
|---|---|---|---|
| $H_2$/CO | $1.455 \times 10^{-14}$ | $8.084 \times 10^{-1}$ | $1.907 \times 10^{-5}$ |
| GRI-Mech 3.0 | $1.567 \times 10^{-14}$ | $1.469 \times 10^{-7}$ | $1.316 \times 10^{-7}$ |
| USC-Mech II | $9.632 \times 10^{-15}$ | $5.567 \times 10^{-3}$ | $1.704 \times 10^{-7}$ |
| $iC_5H_{11}OH$ | $1.227 \times 10^{-10}$ | $1.363 \times 10^{-3}$ | $2.864 \times 10^{-5}$ |

Table D.1: Summary of Jacobian matrix verification results for the Intel OpenCL runtime. The reported error statistics are the maximum filtered relative error $E_{\mathcal{C}}$ and LAPACK error $E_{\mathcal{L}}$ over all vectorization patterns (Table 3.3), CONP/CONV, and sparse/dense Jacobians. The threshold for the filtered relative error is the same as reported in Section 3.3.3.

| Model | $E_{\mathcal{L}}$ | $E_{\mathcal{C}=10^{20}}$ | $E_{\mathcal{C}=10^{15}}$ |
|---|---|---|---|
| $H_2$/CO | $1.456 \times 10^{-14}$ | $1.230 \times 10^{-1}$ | $3.951 \times 10^{-6}$ |
| GRI-Mech 3.0 | $1.014 \times 10^{-14}$ | $1.890 \times 10^{-7}$ | $1.877 \times 10^{-7}$ |
| USC-Mech II | $9.632 \times 10^{-15}$ | $8.998 \times 10^{-4}$ | $1.201 \times 10^{-8}$ |
| $iC_5H_{11}OH$ | $9.133 \times 10^{-15}$ | $1.723 \times 10^{-5}$ | $5.108 \times 10^{-7}$ |

Table D.2: Summary of Jacobian matrix verification results for the Portable OpenCL (POCL) runtime. The reported error statistics are the maximum filtered relative error $E_{\mathcal{C}}$ and LAPACK error $E_{\mathcal{L}}$ over all vectorization patterns (Table 3.3), CONP/CONV, and sparse/dense Jacobians. The threshold for the filtered relative error is the same as reported in Section 3.3.3.

| Model | $E_{\mathcal{L}}$ | $E_{\mathcal{C}=10^{20}}$ | $E_{\mathcal{C}=10^{15}}$ |
|---|---|---|---|
| $H_2$/CO | $5.962 \times 10^{-15}$ | $3.614 \times 10^{-2}$ | $1.657 \times 10^{-6}$ |
| GRI-Mech 3.0 | $1.297 \times 10^{-15}$ | $1.321 \times 10^{-7}$ | $1.316 \times 10^{-7}$ |
| USC-Mech II | $9.630 \times 10^{-15}$ | $4.185 \times 10^{-4}$ | $6.746 \times 10^{-9}$ |
| $iC_5H_{11}OH$ | $6.131 \times 10^{-15}$ | $1.721 \times 10^{-5}$ | $5.108 \times 10^{-7}$ |

Table D.3: Summary of Jacobian matrix verification results for OpenMP execution. The reported error statistics are the maximum filtered relative error $E_{\mathcal{C}}$ and LAPACK error $E_{\mathcal{L}}$ over all vectorization patterns (Table 3.3), CONP/CONV, and sparse/dense Jacobians. The threshold for the filtered relative error is the same as reported in Section 3.3.3.

## D.3 SIMD efficiency scaling example

This simple example demonstrates how the SIMD efficiency of shallow-vectorized OpenCL source-term evaluation depends on the size of the chemical model in question, i.e., the amount of computational work per source-term evaluation. The base chemical model for this example was the isopentanol model [136] used throughout this article, and the same thermochemical state database described in Section 3.3.1 was used for source-term evaluation. As in Section 3.3.5 all reported results are based on 10 individual runs and in this example all cases were run on the `avx2` machine using the Intel OpenCL runtime.

| Model | $E_{\mathcal{L}}$ | $E_{\mathcal{C}=10^{20}}$ | $E_{\mathcal{C}=10^{15}}$ |
|---|---|---|---|
| $H_2$/CO | $1.862 \times 10^{-14}$ | $1.741 \times 10^{0}$ | $4.508 \times 10^{-5}$ |
| GRI-Mech 3.0 | $1.489 \times 10^{-14}$ | $3.842 \times 10^{-7}$ | $3.687 \times 10^{-7}$ |
| USC-Mech II | $1.174 \times 10^{-14}$ | $1.119 \times 10^{-2}$ | $1.983 \times 10^{-7}$ |
| $iC_5H_{11}OH$ | $8.602 \times 10^{-15}$ | $1.748 \times 10^{-5}$ | $5.109 \times 10^{-7}$ |

Table D.4: Summary of Jacobian matrix verification results for Nvidia OpenCL execution. The reported error statistics are the maximum filtered relative error $E_{\mathcal{C}}$ and LAPACK error $E_{\mathcal{L}}$ over all vectorization patterns (Table 3.3), CONP/CONV, and sparse/dense Jacobians. The threshold for the filtered relative error is the same as reported in Section 3.3.3.

**Algorithm 1** A greedy selection algorithm to remove reactions from a base chemical model $M$, while preserving the number of active species.

**Input:** Base chemical model $M$ with reactions $R$ and species $S$

```
function DETERMINE SPECIES COUNT(active)
    for Species S_k in model M do
        species_rxn_count [k] ← 0
        for Reaction R_j in model M do
            if active[j] and (|ν'_{k,j}| + |ν''_{k,j}|) > 0 then
                species_rxn_count[k] ← species_rxn_count[k] + 1
    return species_rxn_count
procedure MODEL GENERATION(M)
    active[j] ← True for all reactions R_j in M
    species_rxn_count ← DETERMINE SPECIES COUNT(active)
    while min (species_rxn_count) ≥ 1 do
        species_rxn_count ← DETERMINE SPECIES COUNT(active)
        for Reaction R_j in model M do
            rxn_count[j] ← min_{∀S_k∈R_j} (species_rxn_count [k])
        remove_at ← argmax (rxn_count)
        active[remove_at] ← False
```

First, the reactions in the isopentanol model were converted to simple reversible Arrhenius reactions by either simply dropping third-body efficiency calculations (third-body enhanced reactions), using the high-pressure-limit coefficients (falloff/chemically-activated and P-Log reactions) or fitting Arrhenius parameters to the calculated rate constant at a fixed pressure (Chebyshev reactions). This conversion made the cost of reaction rate evaluation roughly equivalent between all reactions in model, separating the effect of chemical model size from computational intensities of different reaction types on the SIMD efficiency. Next, a greedy reaction removal algorithm (Algorithm 1) generated a number of models ranging from 2100–186 reactions, in increments of 200 reactions (except the final increment from 200 to 186 reactions).

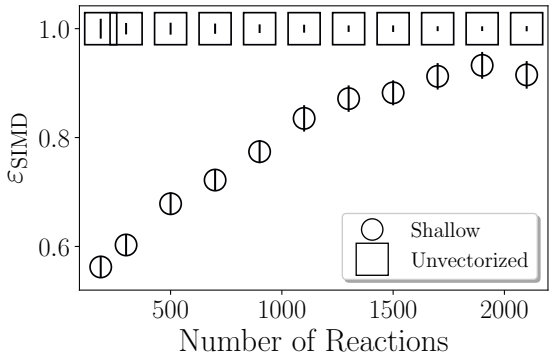



Figure D.1: The effect on SIMD efficiency of varying chemical model sizes.

To discern the effect of varying chemical model size on the SIMD efficiency, shallow-vectorized and unvectorized source-term evaluation performance tests were run. As demonstrated in Fig. D.1 the SIMD efficiency strongly depends on the generated model size and thus the amount of computational work per thermochemical state. In addition, the range of SIMD efficiency in this example (0.56–0.91) is larger than the range of SIMD efficiencies calculated for real chemical models, as seen in Fig. 3.8c. For smaller models—e.g., $H_2$/CO which had a SIMD efficiency of 0.6 on the `avx2` CPU—this suggests that the presence of more computationally intensive falloff/chemically activated reactions in model can increase the SIMD efficiency. However, the base isopentanol model achieved a SIMD efficiency of only 0.78 on the `avx2` machine in Section 3.3.5.1, suggesting that more work could be done to optimize the source-term evaluations. In particular, it is likely that a reaction sorting method, such as suggested by Sewerin and Rigopoulos [53], would be particularly beneficial to reduce the number of vector gather/scatter/masking operations incurred during source-term evaluation.